REASONING ABOUT ACTORS THAT SHARE STATE

SAM CALDWELL

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Khoury College of Computer Sciences
Northeastern University
August 2023
Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Reasoning About Actors that Share State

Thesis Title: ____________________________
Sam Caldwell

Author: ____________________________

X

PhD Program: ___ Computer Science ___ Cybersecurity ___ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Mr. F. G. H. Th. Advisor 7 Aug. 2023

Date

OK

Thesis Reader 8/2/2023

Date

Jlah

Thesis Reader 8/3/2023

Date

Am. Juh

Thesis Reader 8/4/2023

Date

Thesis Reader

KHOURY COLLEGE APPROVAL:

Associate Dean for Graduate Programs 08/11/2023

Date

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Recipient’s Signature 11 August 2023

Date

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.
This dissertation would not be possible without the support of many people, to whom I express my many thanks: I owe a great deal of gratitude to my advisor, Matthias Felleisen, for creating a research landscape to explore, his continual curiosity, and patience. Additionally, discussions and questions from the members of my thesis committee—Jan Vitek, Arjun Guha, and Sam Tobin-Hochstadt—provided valuable insight and helped shape this dissertation.

The research in this dissertation would not exist without the work of Tony Garnock-Jones, who forged a path through the concurrency landscape and then graciously showed me the way. Thank you Tony, I think I will call the cat a #$@*$&!% after all. I have also benefited enormously from many discussions with the members of NU PLT and the PRL, who I thank for sharing their passion for programming languages (and many other interests, besides).

This dissertation research has been partially supported by NSF grants SHF 1763922, CI 1823244, SHF 2116372, and SHF 2315884, plus a gift from Cisco.

My thanks to my family for continuing to cheer me on through the years and not making me squirm too much with difficult graduation-related questions.

Finally, I would not have made it through this process without the unwavering support of The Pack. Sirius and Aiko, from the bottom of my heart, thank you for always being there for me.

— Sam Caldwell
## CONTENTS

1. **The Concurrency Challenge; Actors**
   - 1.1 Thesis 4
   - 1.2 Support and Contributions 4
   - 1.3 Organization 5

2. **The Dataspace Theory of Actors**
   - 2.1 Dataspaces, Informally 7
     - 2.1.1 Racket Dataspaces 7
     - 2.1.2 Putting the Pieces Together 11
     - 2.1.3 The Role of Recursive Interest in Dataspace Protocols 13
     - 2.1.4 The Problems of Programming Actors in Untyped Languages 14
   - 2.2 Dataspaces, Formally 15
   - 2.3 A Streamlined Semantics 19
     - 2.3.1 Metafunctions 21
     - 2.3.2 A Note on the Relationship Between the Semantics 22

3. **Structural Types for Dataspace Actors**
   - 3.1 Types, Informally 23
     - 3.1.1 Typing the Chat Room 30
     - 3.1.2 Revisiting the Arithmetic Service 32
   - 3.2 Actors, Types, Formally 33
     - 3.2.1 \( \lambda_{ds} \) Reduction Metafunctions 35
     - 3.2.2 Types for Dataspaces and Actors 38
     - 3.2.3 Static Semantics 39
     - 3.2.4 Meta-Theorems 47
   - 3.3 Completing the Model 48
     - 3.3.1 Meta-Theorems 50
   - 3.4 Implementation 53
   - 3.5 Assessment 55
   - 3.6 Related Work 56

4. **Facets for Actors that Share State**
   - 4.1 Facets, Informally 59
     - 4.1.1 Derived Forms 64
     - 4.1.2 The Synergy of Facets 66
   - 4.2 Facets, Formally 67
     - 4.2.1 Formal Syntax 67
     - 4.2.2 Semantics 70
     - 4.2.3 Metafunctions 74
     - 4.2.4 Machine Transition Example 79

vii
4.2.5  Dataspace Programs  82
4.3  Type Checking Facets  83
  4.3.1  Auxiliary Definitions  85
  4.3.2  Machine Typing  87
4.4  Properties  89
4.5  Related Work  92
5  Behavioral Types for Faceted Actors  95
  5.1  Overview  95
    5.1.1  Basic Types  95
    5.1.2  Specification  95
    5.1.3  Effect Types for Facets  97
    5.1.4  Checking Specifications Mechanically  98
    5.1.5  Revisiting the Smart-Home Example  99
  5.2  Type Level Semantics  100
    5.2.1  Machine Types and Type-level Machines  103
    5.2.2  Type-level Dataspaces  107
  5.3  Meta-Theorems  107
  5.4  Implementation  113
    5.4.1  Implementing the Behavior of Facet-Oriented Actors  113
    5.4.2  Implementing the Type Checker  114
    5.4.3  Implementing the Model Checker Translation  120
    5.4.4  Type-Varying Assertions  125
  5.5  Evaluation  126
    5.5.1  Corpus  127
    5.5.2  Deadlock Freedom  132
    5.5.3  Classifications  134
    5.5.4  Results  136
    5.5.5  Interpretation and Discussion  139
    5.5.6  Threats to Validity  141
  5.6  Related Work  141
6  Conclusion  145
  6.1  Future Work  145
  6.2  Conclusion  149
A concurrent program consists of some number of independently running components. The behavior of the individual components and their interactions define the behavior of the program as a whole. As the number of components and the complexity of their interactions increases, the number of different behaviors a program may exhibit for the same input explodes exponentially. Reasoning about such programs is a major challenge for developers.

Concurrency arises in programs due to a need to juggle multiple tasks inherent in the problem domain. For example, a GUI application needs to remain responsive to user inputs while working on other tasks, such as querying a web server. Concurrency may also be motivated by efficiency. Organizing a task, or tasks, as a concurrent program may allow for utilizing system resources, such as CPUs, in a manner that may not be possible for sequential programs.

The reality, then, is that concurrent programming is both necessary and incredibly challenging.

In response, researchers and professionals have pursued numerous ideas for imposing order on the landscape. These ideas range from organizational methodologies such as design patterns to reimagining the foundational substrate of computation.

Along this latter direction, numerous theories of concurrent programming provide developers with alternate approaches to organizing concurrent components and orchestrating their interactions. In this context, a theory is a definition of a concurrent component together with the mechanisms and rules governing interaction. The expectation is that the organizing principles provided by a concurrency theory offer developers a good way to design and understand programs.

In general, the differences between the concurrency theories manifest in their internal organization rather than external behavior. That is, most theories enable the implementation of the same black-box system behaviors. For example, researchers have shown the duality between multi-threaded and event-driven programming and the interchangeability of different message-passing styles. Internally, however, the theories provide significantly different guarantees, reasoning capabilities, and organizational guidelines. The task, then, in designing concurrency theories is choosing which aspects to emphasize for ease of use and reasoning, and which to sideline.

Over the past two decades, a number of programming languages have added libraries for programming with the Actor theory. This theory of concurrency sports an easy-to-use mechanism for expressing concurrent computation. A program spawns actors, and those converse via messages toward a common specification. Like
conversations of a large dinner party, these conversations can run in parallel and in a nested fashion.

The choice of message-sending aims to reduce the potential for problems from the thread-based world, most importantly deadlock and data races. It facilitates stating expectations about a concurrent program and validating them. Given the enormous difficulties of concurrent programming, these advantages should encourage programmers to embrace actor-based libraries and the underlying theory.

A corpus investigation [99] of actively used actor programming libraries suggests a different picture, however. When a language supports both message-sending actors and shared-state threads, programmers often mix and match the two modes of programming. Follow-up interviews confirm that programmers tend to point to the ease of sharing among concurrent components and to performance improvements. Their actor-based components communicate via messages when possible, but they share state when doing so seems safe and is convenient. In short, programmers wish for some amount of sharing among actors.

Structural and substructural type systems are another widespread tool for reasoning about programs [19, 66, 70]. Equipping a language with a type system can make use of such reasoning capabilities. Generally speaking, the purpose of a type system is to help programmers check statements about their code and to assist them with designing programs. For example, a structural type system checks constraints on the kind of values a function may consume. If a function’s argument type is int, all argument expressions must have type int, meaning the function will never have to process any other kind of value. In turn, the programmer can rely on this “theorem” when designing the function’s body. Hence a structural type system also assists programmers with the systematic design of programs. Substructural types can impose and check additional constraints out of reach for structural systems. Rust’s ownership types are an example: only one thread at a time may own and modify a mutable data structure.

Especially in the world of concurrency, researchers have gone beyond structural types and have investigated behavioral type systems, which describe and enforce properties of the program across distinct steps in its execution [57, 60]. A behavioral type system generalizes the expressible constraints. Instead of individual function calls, a behavioral type system may be able to express and check constraints on a series of calls to the same function. For example, sophisticated GUI frameworks demand that clients call methods in a particular order so that the initial state is properly set up and functions as promised.

In applying behavioral type systems to concurrency theories, researchers have found the \( \pi \)-calculus particularly fruitful [57, 58, 60]. Efforts to apply the ideas in other settings have found less success [22, 76]. This speaks to a truism for all static analyses: the same behavior, programmed in different styles, may be more or less amenable to automated reasoning. A concurrent language theory, then, may be more amenable to behavioral type system, by enforcing a reasonable program structure.
Some developers find message passing easy to reason about. Processes and actors are nice concrete components. Message sending allows causal reasoning about what happens and why in the program. Within the world of message passing, much of the attention has been focused on point-to-point communication. Point-to-point is great for many situations, but there are also many times where components must organize and communicate within groups. Such instances are awkward to express.

The dataspace theory of actors addresses this need for sharing \cite{13, 41, 42}. It supplements the actor theory with a space for shared state. Conceptually, it accepts the actor theory’s premise that computation is conversation but embraces the idea that conversations build up shared context, which deserves explicit representation.

In the dataspace theory, actors deposit facts—dubbed assertions—into a shared dataspace to set up context and remove them when they are no longer needed. Conversely, an actor tells the dataspace of its interests in certain kinds of assertions. The dataspace matches assertions and interests and informs actors about such matches—and changes to them. Technically, the dataspace calls the actor and waits for a response, a series of actions to be executed on behalf of the actor. From the dataspace’s point of view, an actor is thus a function that reacts to changes in the published states of other actors. Hence, programming individual actors functionally seems a natural choice; in particular, functional code often lends itself to simple reasoning \cite{59}.

In terms of type-based reasoning about dataspace programs, the first challenge is to incorporate dataspace programming into a statically typed programming language. Type systems, by rule, are intimately interwoven with the language’s mechanism for moving and communicating data. As a novel communication mechanism, traditional type systems need to be extended in some way in order to have any control about what goes in and comes out of a dataspace. With such control, a programmer can utilize types and be sure that the structure of communications, that is, the contents of assertions, is consistent across the components in the program.

Beyond structural types, the question is whether the dataspace theory is amenable to the definition of a behavioral type system. Adapting this idea to the world of (dataspace) actors a programmer may ask—and may wish to check with a behavioral type system—whether a collection of actors controlling a home-lighting system satisfy constraints such as the following:

\begin{itemize}
  \item It is always true that (1) if a room contains at least one person and the wall switch is on, the light eventually comes on and (2) if a room is empty, the light is eventually off.
\end{itemize}

The italicized words in this statement indicate that statements about the dynamic behavior of an actor system call for a temporal logic. The challenge is to design a behavioral type system that can verify such statements.

While the dataspace theory provides some abstraction, programming such actors with plain functional code tends to conflate separate conversations and, within a
conversation, different reactions to changing assertions. Thus a static analysis (types or otherwise) cannot feasibly reconstruct the shape and flow of assertions as the conversation evolves, let alone whether a composition of state-sharing actors achieves its specified goal.

Garnock-Jones’ proposal of a *facet* notation—a domain-specific language (DSL) for programming individual actors—works backwards from the concepts of the dataspace to explicate the structure of an actor’s participation in conversations [40]. That is, it comes with linguistic constructs that express when and how an actor reacts to changes in the dataspace and when it adds or removes its assertions. Similarly, it includes constructs to delineate nested conversations. Imposing such constructs on the language of individual actors—and making them second class—shifts the trade-off from expressive power toward reasoning capability [31]. Technically, it trades some of the power of a functional language for automatic inferences about the communication among a group of dataspace actors.

1.1 Thesis

*A domain-specific language for programming dataspace actors enables type-based reasoning and automatic verification of program behavior.*

1.2 Support and Contributions

My dissertation supports the thesis with the following contributions:

- **A Structural Type System for the Dataspace Theory.** This work demonstrates that dataspace communication can be reconciled with static type checking without resorting to a unityped interface. This allows for dataspace programs to enjoy the traditional benefits of types: such as design, documentation, optimization, error detection, and so on. I present a prototype implementation of the type system for the Racket dataspace library. This type system ensures that when actors state an interest in certain shapes of assertions, they can constrain the kind of values inside of the assertions. Conversely, programmers can design actor functions assuming the nature of value exchanges. In support I prove that the type judgment satisfies key soundness and termination properties. The semantics and theorems give rise to the first complete semantics for dataspace programs.

- **A Structural Type System for the Facet DSL.** The structural type system is for a dataspace library that provides a procedural interface for actor behaviors. That is, the library provides data structures for representing concepts such as assertions and actors and functions for manipulating them, and otherwise relies on general purpose procedures for weaving them together as actor behaviors. However, a
procedural interface imposes a black-box interface for actors, hindering behavioral reasoning. The facet DSL provides a structured notation for dataspace actor behaviors that has the potential for enabling behavioral checks. Hence, the differences with the procedural notation must be reconciled with respect to structural checking. In this work, I show that the static type system is not specific to a particular notation for describing dataspace actors but generalizes to others such as the facet DSL. On the practical side, I present a prototype of the typed facet language. On the theory side, this dissertation includes an adaptation of the structural type system for facets and its soundness theorem.

- *An Analysis of Type-Based Behavioral Properties of Faceted Actors.* Finally, I present a tool for automatically checking behavioral properties of dataspace programs. This demonstrates that the dataspace theory, especially when using the facet DSL, is amenable to type-based reasoning about communication in a behavioral sense. A compiler gathers behavioral types from the typed facet implementation and translates these into a program for an off-the-shelf model checker. The evaluation of this approach reports results of applying it to a corpus of dataspace programs. A formal semantics of facets, dataspaces, and types that allows comparing the behavior of programs and type-based abstractions of those programs provides a theoretical foundation. The key theorem shows that specifications describing temporal properties of assertion types transfer from the type to the term level.

1.3 ORGANIZATION

This dissertation is organized as follows.

Chapter 2 is an overview of the dataspace model of actors, providing the basis for the rest of the chapters. Starting with an informal overview, it uses examples from a Racket dataspace library to illustrate this programming style. The examples set up a discussion of some of the appeal of dataspaces. The initial formal view of the model is agnostic with respect to the language for implementing each individual actor. The behavior of each individual actor is simply a function. The formal syntax and semantics can then be formulated with a focus on the essence of the system-level transition system for dataspace-program states. The chapter provides two such semantics. One breaks down the progress of dataspace programs into fine-grained states. This model is useful as the basis for implementation. Another, alternative, semantics streamlines the first model to focus on communication between dataspace actors. This compact model is useful for formally analyzing properties of dataspace communication in chapter 5.

Chapter 3 explores the challenge of using a typed language for programming dataspace actors. It then shows the design of a typed language that incorporates knowledge of dataspace communication, yielding a sound structural type system. The examples in the chapter are written using the prototype implementation of the type system. They
set up a discussion of the benefits of typed dataspace programming as well as the limitations of such a structural design compared to what may be desired by programmers. The implementation of the prototype uses the recently invented “types-as-macros” technique [18], which warrants further discussion. A formalization of the semantics for programming typed dataspace actors provides the theoretical basis of the chapter. Additionally, the chapter introduces notions needed to formally link the semantics of individual actors with the semantics of dataspace programs from chapter 2. With such notions, it becomes possible to prove both the soundness of the individual actor language as well as complete dataspace systems.

The focus of chapter 4 is the DSL of facets for programming individual dataspace actor behaviors. Examples from the actual implementation informally illustrate the language features. A formal semantics assigns precise meaning to terms in the language. An adaptation of the structural type system from chapter 3 as a type-and-effect system bridges the differences of the facet notation and is proven sound.

Chapter 5 shows an approach to utilizing the structure of the facet DSL to check behavioral properties of dataspace programs. The idea is first explained and demonstrated informally. Then, a formulation of type-level behavior is formalized and used to prove the metaproperties that validate the approach. The chapter discusses the primary aspects of the implementation, including a number of features for supporting real programs and facilitating ease of use. An evaluation of the behavioral checker demonstrates the usefulness of this approach and discusses a number of aspects of the experience.

Finally, chapter 6 concludes by reflecting on the lessons and insights from this experience and proposing future work.
THE DATASPACE THEORY OF ACTORS

2.1 DATASPACES, INFORMALLY

The dataspace theory of actors is parameterized over a base language for programming each individual actor. For an introduction using illustrative examples it is still best to use some concrete syntax. This section uses Racket syntax, falling back on an untyped prototype implementation. This section provides an overview of the basic concepts (section 2.1.1), illustrates it with examples (sections 2.1.2 and 2.1.3), and makes the case for introducing a structural type system (section 2.1.4).

2.1.1 Racket Dataspaces

In the dataspace theory, an individual actor combines a private state with a behavior function. The private state is any piece of data from the underlying language. In a chat server program, for example, a room-list may have the shape

\[
\text{(list "FBI" "CIA")}
\]

recording the name of each active chat room.

Intuitively, an actor behaves as if it were a function that maps events and the actor’s current, private state value to a new state value and some instructions to carry out on its behalf:

\[
\text{Event } \times \text{ State } \rightarrow \text{ State } \times \text{ Instructions}
\]

The connecting dataspace layer interprets instructions. Essentially, the goal of the dataspace is to enable conversation among actors and manage their state.

A dataspace equips a group of actors with a means of aggregating and sharing items of information, called assertions. The chat server dataspace creates one actor for each connection to a user. This user-agent actor is responsible for relaying chat messages and carrying out commands from the remote user. For example, a connected user under the alias “Mo” may issue a request to join the room named “FBI”. In response, the user’s agent actor states the following assertion:

\[
\text{(in-room "Mo" "FBI")}
\]

In the syntax of the Racket prototype, this assertion is stated as a Racket struct form. Racket structures are fixed-length tuples declared by the programmer, where the name (in-room) serves as an identifying tag. When the user leaves the room, the user-agent
withdraws this assertion. Assertions range over basic values—numbers, strings, and so on—as well as first-order, immutable data structures, including lists and programspecific structures. Assertions have the same status in dataspace programs as facts in a Prolog database [20].

The dataspace model links an actor with its assertions in a particular way. An assertion is read-only, and only the actor making an assertion may remove it. While multiple actors may make identical assertions, dataspaces hide such redundancy by providing a set view of an underlying bag of assertions. Additionally, the lifetime of an actor bounds the lifetime of its assertions. When an actor terminates or crashes, the dataspace removes its current assertions.

In order to observe the appearance and disappearance of assertions, an actor makes an assertion of interest. An assertion expressing interest in the above chat-room assertion uses the built-in structure observe (figure 2.2 summarizes the extensions to base Racket):

```
(observe (in-room "Mo" "FBI"))
```

This particular assertion of interest could originate from an actor managing the room list. The dataspace notifies this actor—and any other actor expressing this interest—when the user-agent asserts `(in-room "Mo" "FBI")`, and again when it withdraws the assertion.

Actors often assert interest in entire families of related assertions. For example, the room-list actor tracks the presence of every user in the "FBI" room, not just "Mo". To express interest in all such values, the actor uses the wildcard `⋆` to assert

```
(observe (in-room ⋆ "FBI"))
```

In response, the actor receives a notification each time a matching `in-room` assertion appears or disappears. Semantically, a wildcard creates infinite sets of assertions; routing of notifications can thus be understood in terms of set intersection.

Figure 2.1 provides a visual intuition for the chat server dataspace described thus far. The "Mo" user has recently joined the "FBI" room. Accordingly, the user-agent actor places an `in-room` assertion in the dataspace. Because the room-list actor’s assertions include an expressed interest in `in-room` assertions, the dataspace routing mechanism recognizes the overlap and sends an event.

An expression of interest is itself an assertion, observable by other actors. A common use of this recursion concerns demand matching. An actor capable of matching demand for some resource or information `c` asserts `(observe (observe c))`, while actors interested in `c` assert `(observe c)`. The dataspace routing mechanism duly informs the first actor of the interest. In response, it performs the computation to produce the assertion `c`. Finally, dataspace routing again kicks in to notify the interested actor in the appearance of `c`.

8
Actors receive notification \textit{Events} regarding changes to the dataspace’s assertions in the form of a patch, which contains two sets. The first is a set of assertions newly \textit{added} to the dataspace matching the actor’s expressed assertions of interest. The second is a set of relevant assertions that have been \textit{removed} from the dataspace. Hence, the room-list actor receives the notification

\[
(\text{patch } \{(\text{in-room "Mo" "FBI"})\} \emptyset)
\]

when user "Mo" joins. When the user leaves the room the actor receives the corresponding notification \( (\text{patch } \emptyset \{(\text{in-room "Mo" "FBI"})\}) \).\footnote{Dataspaces additionally support \textit{message broadcast}, though this presentation focuses on communication through assertions. Messages may be thought of as a special case of assertion-based communication: sending a message atomically combines the assertion and immediate retraction of its value, i.e., \((\text{message } c)\) behaves roughly as \((\text{patch } c \emptyset)\) atomically followed by \((\text{patch } \emptyset c)\).}

Expressions between curly braces \{e \ldots\} create sets of assertions while actors analyze incoming sets using \textit{project}, which resembles set comprehension. Projection takes the form \( (\text{project [pattern set] body}) \), as in

\[
(\text{project } [[\text{in-room $name$ "FBI"}]) \{(\text{in-room "Mo" "FBI"})\}] name)
\]

The pattern serves to filter and destructure assertions in \textit{set}. Evaluation instantiates the body of the \textit{project} form once for each assertion in the set matching the given pattern, yielding a list. Here, it yields \((\text{list "Mo"})\). In addition to binding variables, prefixed with $\$, patterns may include the discard symbol \_ which matches any value (without providing a binding). Since assertion sets may be conceptually infinite, some \textit{project} expressions may yield infinite lists, as in \((\text{project } [$x \{\_\}] x)\). The implementation signals an error when this happens.\footnote{An alternative interpretation is that of a diverging computation, instantiating \(x\) an infinite number of times. Since these infinite sets are recognizable, the implementation opts to signal an error.} The discard pattern \_ is especially useful for
Launch a dataspace program with some initial actors

An actor with a behavior plus initial state and assertions

Create a set of assertions

An assertion of interest

Describes all possible assertions

Create a patch of two assertion sets

Update private state plus some instructions to perform

Termination with some final instructions

Iterate over the matching assertions in a set

Pattern that matches anything and binds it to \( name \)

Pattern that matches anything

Figure 2.2: Extensions to Racket for Dataspace Programming

analyzing such sets, as it can ignore infinite portions and allow the rest of the pattern to successfully match. For example, the expression \((\text{project } [\_ \{\star\}] \text{ "match"})\) yields the value \((\text{list } \text{"match"})\).

Actor behavior functions actually return values belonging to a disjoint sum, as opposed to the simplistic product described above. The sum allows an actor to shutdown in an orderly, non-exceptional manner. When an actor has fulfilled its purpose, it submits the first form of response, a \(\text{quit} \) record, to the dataspace:

\[(\text{quit } \text{instructions})\]

In response, the dataspace removes the actor, withdrawing each of its assertions, and carries out some final \(\text{instructions}\) on its behalf. In the course of processing a notification, an actor may raise an exception. The dataspace layer interprets uncaught exceptions as \((\text{quit } \text{empty})\) instructions.

The second form of return value makes or withdraws assertions via \(\text{patch}\)-es or augments the program with an entire new actor. In either case, the actor submits

\[(\text{transition } \text{state } \text{instructions})\]
Such a transition record provides a new, updated private state for the actor and a list of instructions for the dataspace to carry out, in order. An instruction might be a simple patch. For example, the user represented by the "Mo" actor may issue a request to switch to the "CIA" chat room. To this end, the "Mo" actor would update both its in-room assertion, notifying other actors of the change, and its assertion of interest, allowing it to monitor other users present in the new room. It accomplishes this goal with this transition:

```
(transition
  "CIA"
  (list
    (patch {{(in-room "Mo" "CIA") (observe (in-room "CIA"))}}
          {{(in-room "Mo" "FBI") (observe (in-room "FBI"))}})))
```

The first item updates the actor’s private state value to "CIA", recording the user’s current room. The transition provides a single instruction, a patch manipulating the actor’s assertions. The first set in the patch places the assertion (in-room "Mo" "CIA") in the dataspace as well as an assertion of interest in other users in the "CIA" room. The second set in the patch withdraws the corresponding assertion of presence and interest for the previous "FBI" room.

An actor might also request the creation of a new actor. A corresponding instruction is an actor specification, which takes the form

```
(actor behavior state assertions)
```

providing a description of the actor’s behavior function as well as its initial private state and assertions.

Finally, the form

```
(dataspace actors)
```

launches a dataspace program with the given initial actors. Figure 2.2 summarizes the extensions to Racket for dataspace programming.

2.1.2 Putting the Pieces Together

Figure 2.3 shows a sketch of the user-agent actor implementation for a chat room, plus some of the context. The main function launches a dataspace with the described user-agent, room-list, and any other potential actors. The actor’s initial state establishes the presence of user "Mo" in room "FBI", as described above. The behavior of the actor analyzes patches from the dataspace, projecting out the first slot from in-room assertions. The actor creates a user-notification for each arriving and departing peer. Another actor, not depicted, transforms such notifications into suitable network messages.
(define (main)
  (dataspace
   (list
    (create-user-agent "Mo" "FBI")
    (create-room-list ...)
    ...))))

(define (create-user-agent name initial-room)
  (actor
    ;; behavior: notify the user as peers enter and
    ;; leave the current room.
    (lambda (event current-room)
      (define arrivals
        (announce (patch-added event) current-room " arrived."))
      (define departures
        (announce (patch-removed event) current-room " departed."))
      (transition current-room
        (append arrivals departures)))
    initial-room
    {{(in-room name initial-room)
      (observe (in-room ⋆ initial-room))}})

;; Create user-notification patches based on
;; presence assertions in an event.
(define (announce assertions current-room message)
  (project [{(in-room $who current-room) assertions}
    {patch {(user-notification (string-append who message))} {}}}))

Figure 2.3: A user-agent actor

HIGHLIGHTING DATASPACES  The chat example illustrates several benefits of dataspase coordination over pure message-passing. Garnock-Jones [40] surveys the comparative advantage of dataspaces versus other concurrency theories in greater depth, using different implementations of a similar chat example.

• The temporal duration of assertions, i.e. the fact that an assertion remains in the dataspase until the originating actor withdraws it or terminates, allows new actors to seamlessly join an ongoing conversation. When a new user connects to the chat server, the program spawns a new user-agent actor on its behalf. This new actor expresses an interest in other members present in the current room. The interest effectively acts as a query of existing matching assertions. In response, the dataspase provides a description of all members present in the room.

• After joining a room and learning of existing members, the same assertion of interest provides the user-agent with ongoing incremental knowledge updates. The
actor receives a notification each time a current user leaves or a new user joins the room.

- **Group communication** simplifies the protocol and implementation of presence information. The user-agent actor makes a single assertion that informs every peer to whom it is relevant. Assertions decouple actor identities. They allow communication without knowing exactly which actor(s), or even how many, to address. In traditional point-to-point models, a designated actor must maintain a collection of addresses or channels to enable communication among the other actors. As Joe Armstrong, designer of Erlang points out [6, §8.7], “If we want to send a message to a process, then we need to know its PID. This is often inconvenient since the PID has to be sent to all processes in the system that want to communicate with this process.” In the same book (chapter 11), a similar multi-room chat example requires an additional actor for each possible room to keep track of this information.

- The **link between failure and communication** provides a uniform and convenient method for resource management in the presence of faulty actors. In this example, the ability to communicate with a particular chat user may be considered as a resource. As the availability of this resource is announced via assertion, it is automatically tied to the lifetime of the corresponding actor. An exception in the actor results in the withdrawal of the assertion, notifying interested peers. From the perspective of other user-agents, both orderly and exceptional termination convey the same information—that a certain peer is no longer communicable. Other actors in the system, meanwhile, can react in different ways, perhaps to clean up any associated network connection(s).

### 2.1.3 The Role of Recursive Interest in Dataspace Protocols

Dataspace protocols benefit from the use of recursive subscriptions, that is, assertions of interest in assertions of interest (and so on).

To illustrate the point, consider an “arithmetic service” dataspace. Assertions take the form \((\text{sum } x \ y \ z)\) where \(z = x + y\). Clients of the service make requests by asserting interest in sum assertions with particular numbers for \(x\) and \(y\), such as

\[
(\text{observe } (\text{sum } 4 \ 5 \ *))
\]

The actor implementing the service listens for requests with

\[
(\text{observe } (\text{observe } (\text{sum } \ * \ * \ *)))
\]

in this case responding \((\text{sum } 4 \ 5 \ 9)\). The arithmetic service answers requests by analyzing the \(x\) and \(y\) values of sum requests, ignoring \(z\):

\[
(\text{project } ((\text{observe } (\text{sum } $x \ $y \ _)) \ e) \ (\text{sum } x \ y \ (+ \ x \ y)))
\]
The example illustrates how interpreting the assertion of interest as a request yields the simplest possible protocol. Abstractly, this protocol for request/response situations combines the request assertion with the interest in the result. It thus reduces both the number of assertions an actor must state to make a request and the different types of assertions comprising the conversation. To appreciate the protocol’s simplicity, consider the next-best alternative, which would be to use two distinct structs. One struct would represent requests separately via a \( (\text{sum-request } x y) \) assertion. The other would describe the results of requests as a \( (\text{sum } x y z) \) assertion. Assertions of interest in results must include the inputs \( x \) and \( y \), so that there is no confusion with answers to other requests. This requirement is equivalent to \( (\text{observe } (\text{sum } x y *)) \), which is the request from above and suggests the simplification to a single struct.

2.1.4 The Problems of Programming Actors in Untyped Languages

Not surprisingly, using an untyped functional language as the implementation substrate of dataspace actors leads to serious problems.

(1) Productive communication between actors requires an agreement on the kinds of data to be exchanged; programming in such a manner is analogous to following a type discipline. Untyped languages like the Racket prototype provide little help with finding, and no help adhering to, a type structure that governs the communication between components. Consequently, the potential for mismatched messages undermines the principle of failure isolation. A buggy actor may make utterances that violate these expectations, but it is the actors that interpret the faulty messages that suffer the consequences. Worse, some of these mistakes correspond to simple type errors. A bug in a user-agent actor may turn a numeric user input into an assertion about its name, as a number rather than a string, causing other actors to crash while merrily continuing its own existence.

(2) The recursive nature of (interest in) assertions complicates the matter further. In the arithmetic dataspace from section 2.1.3, a logic programmer may scan the description of sum assertions and surmise that the protocol lends itself to computing differences as well as sums, as with \( (\text{observe } (\text{sum } 4 *) ) \). Such a request is incompatible with the implementation of the arithmetic service described above; it violates the assumption that the \( y \) slot is a number, even though it matches the service’s assertion of interest \( (\text{observe } (\text{observe } (\text{sum } *) *)) \). As discussed in section 2.1.1, project errors when a pattern variable has an infinite number of matches, so delivering such an assertion to the service actor causes it to crash—another failure to isolate misbehaving actors.

(3) The principle of scalability requires a certain amount of cooperation between actors in terms of execution time. Indeed, Garnock-Jones et al. [41] show several key properties of the dataspace model under the assumption that the behavior function of every actor is total (including exceptions). However, programming actors in an untyped
functional language such as Racket or ECMAScript allows the creation of both total and partial behavior functions, with no way to distinguish the two. In particular, divergence is a form of failure, but it bypasses the reasoning mechanism provided by the language for these situations—automatic withdrawal of assertions. The termination guarantees of a simple type system aid in isolating and responding to failures as expected in the actor model.

(4) Additionally, developers may wish to place specific restrictions on the behavior of certain actors. For instance, in the case that some rooms are private, the developer of the chat server may wish to ensure that user-agent actors do not make overly broad queries. Permitting such queries would allow a curious user to discover the existence of, and join, private channels without an invitation. Types are one mechanism for imposing such constraints. If the type of user-agent actors indicates that no such subscriptions are made, the protocol is safe from intrusions.

2.2 Dataspaces, Formally

Figure 2.4 provides a syntax for dataspace systems. The internal behavior of each actor is a mathematical function. This interface is an abstraction point; it separates the concerns of an individual actor implementation from the details of dataspace concurrency and coordination. Consequently, it is just as easy to use the dataspace configuration syntax and semantics with a model of a Racket-like functional language as with a model based on ECMAScript [29].

\[
\begin{align*}
\text{Programs} & \quad P \in \text{Prog} = \text{dataspace } S \\
\text{Actor Specifications} & \quad S \in \text{Specs} = \text{actor } f v \pi \\
\text{Behavior Functions} & \quad f \in \text{BehFun} = \exists \text{State } \times \text{State } \rightarrow \text{total } \text{State } \times \text{Action} \\
\text{Events} & \quad e \in \text{Event} = \Delta \\
\text{Actions} & \quad a \in \text{Action} = \Delta \mid S \\
\text{Patches} & \quad \Delta \in \text{Patch} = \pi^+ / \pi^- \quad \text{where } \pi^+ \cap \pi^- = \emptyset \\
\text{Errors} & \quad \dagger \in \text{Error} = \text{error}_{\text{ds}} \\
\text{Assertion sets} & \quad \pi \in \text{ASet} = \mathcal{P}(\mathcal{C}) \\
\text{Assertions} & \quad c \in \text{Assertion} = b \\
& \quad | \text{observe } c \\
& \quad | m(\overline{c}) \\
\text{Basic Values} & \quad b \in \text{BasicVal} = \ldots \text{ base values}
\end{align*}
\]

Figure 2.4: Syntax of Dataspaces

Writing down the reduction semantics for dataspace programs—modulo the semantics for internal actor behavior—requires three elements: (1) a generalized syntax to
specify intermediate configurations; (2) several basic notions of reduction; and (3) metafunctions to keep the formulation of the reductions concise.

**Evaluation Syntax.** Figure 2.5 defines the evaluation syntax of dataspace programs. A running dataspace configuration $C$ contains a pending action queue, the set of all current assertions, and the contained actors. Each actor is represented by an internal name $\ell$ and a state $\Sigma$. The state of an actor consists of a queue of events to handle, a behavior $B$ comprised of a function and current private state value $(f, v)$, and a queue of actions that need to be processed by the surrounding dataspace. Assertions consist of basic values, such as strings or numbers, as well as assertions of interest (observe $c$) and labeled facts ($m(\overrightarrow{c})$). Quiescent terms are those without pending actions; inert terms have neither pending actions nor events to handle.

**Figure 2.5.** Evaluation Syntax and Inert and Quiescent Terms of Dataspaces

**Reduction Relation.** Figure 2.6 presents the reduction semantics of dataspaces. The $\rightarrow_{\Sigma}$ relation operates on individual actor states $\Sigma$, while the $\rightarrow_{ds}$ relation describes the reduction of dataspace configurations, with the aim of reaching quiescent or inert states:

- notify hands an event to the behavior function of an actor and records the new state plus ensuing actions;
- exception terminates an actor that raises an exception;
• spawn creates a new actor;
• gather enqueues an action for the dataspace;
• patch realizes a state-change notification; and
• schedule selects which actor to run.

\[ \delta e_0 \triangleright (f, v) \triangleright \overrightarrow{a} \rightarrow_{\Sigma} \delta \triangleright (f, v') \triangleright \overrightarrow{a'} \overrightarrow{d} \]

when \( f(e_0, v) = (\overrightarrow{a'}, v') \)

\[ \delta e_0 \triangleright (f, v) \triangleright \overrightarrow{a} \rightarrow_{\Sigma} \lambda e. (\cdot, v, v) \triangleright \emptyset \overrightarrow{a} \]

when \( f(e_0, v) \in \text{Error} \)

\[ [([k', a](k, S); R; \overrightarrow{A_Q}) \rightarrow_{ds} [(k', a); R; \overrightarrow{A_Q}(\ell \mapsto \text{boot}(S))] \]

where \( \ell \) distinct from \( k \), every \( k' \),
and the labels of every \( A_Q \)

\[ [([k, a]; R; \overrightarrow{A_Q}(A_{\text{out}}(a')) \overrightarrow{A}) \rightarrow_{ds} [(\ell, a''); (k, a); R; \overrightarrow{A_Q}(\cdot) \overrightarrow{A}] \]

where \( A_{\text{out}}(a) = \ell \mapsto \overrightarrow{a'} \triangleright B \triangleright \overrightarrow{a''} \)

\[ \Delta' = (\pi^+ - \{ c \mid (k, c) \in R \}) \]

\[ \left(\pi^- \cap \{ c \mid (k, c) \in R \}\right) \]

\[ E^\Sigma[\Sigma_Q] \rightarrow_{ds} \text{round-robin}(E^\Sigma, \Sigma') \quad \text{if} \quad \Sigma_Q \rightarrow_{\Sigma} \Sigma' \]

Figure 2.6: Reduction Semantics of Dataspaces

**Metafunctions.** Figure 2.7 defines \( bc\Delta \), the metafunction that implements routing and the semantics of observe. It computes the relevant changes to an actor with label \( \ell \) based on a patch made by actor with label \( k \). In the case that \( \ell = k \), it must also consider the possibility that the actor’s interests change as a result of the patch. In that case, the constructed patch \( \Delta\overrightarrow{fb} \) reflects the most up-to-date interests of the actor.

The following metafunction definitions complete the reduction semantics of dataspaces.
\[ \mathbf{bc}_\Delta : \text{Address} \times \text{Event} \times \text{Table} \times \text{Actor}_Q \rightarrow \text{Actor} \]

\[ \mathbf{bc}_\Delta(k, \Delta, R^{\text{old}}, A) = \begin{cases} \ell \mapsto \Delta_f \triangleright B \triangleright & \text{if } \ell = k \text{ and } \Delta_f \neq \emptyset/\emptyset \\ \ell \mapsto \Delta_{\text{other}} \triangleright B \triangleright & \text{if } \ell \neq k \text{ and } \Delta_{\text{other}} \neq \emptyset/\emptyset \\ \ell \mapsto \triangleright B \triangleright & \text{otherwise} \end{cases} \]

where

\[ \Delta = \pi_{\text{add}} / \pi_{\text{del}} \]
\[ A = \ell \mapsto \triangleright B \triangleright \]
\[ \Delta_f = \{ c | (\ell, c) \in R^{\text{old}}, (f, \text{observe } c) \in R^{\text{new}} \} \cup \{ c | c \in \pi_{\text{add}} \} \cup \{ c | (\ell, c) \in \pi_{\text{add}} \} \]
\[ \Delta_{\text{other}} = \{ c | (\ell, c) \in R^{\text{old}}, j \neq k \} \]
\[ R^{\text{new}} = R^{\text{old}} \oplus (\ell, \pi_{\text{add}} / \pi_{\text{del}}) \]
\[ \pi_{\text{add}} = \pi_{\text{add}} - \pi^\bullet \]
\[ \pi_{\text{o}} = \{ c | (j, c) \in R^{\text{old}} \} \]
\[ \pi_{\text{del}} = \pi_{\text{del}} - \pi^\bullet \]

Figure 2.7: Dataspace Routing

**Definition 1.** The boot function initializes an actor or configuration from an initial specification:

\[ \text{boot} : \text{Specs} + \text{Prog} \rightarrow \text{ActorState} + \text{Config} \]

\[ \text{boot} \left( \text{actor } f \circ \pi \right) = \cdot \triangleright (f, v) \triangleright \pi / \emptyset \]

\[ \text{boot} \left( \text{dataspace } \hat{S} \right) = \left[ (0, S); \emptyset; \cdot \right] \]

**Definition 2.** The \( \oplus \) operation incorporates a patch into a dataspace’s stored knowledge:

\[ \oplus : \text{Table} \times (\text{Address} \times \text{Patch}) \rightarrow \text{Table} \]

\[ R \oplus (k, \pi_{\text{add}} / \pi_{\text{del}}) = R \cup \{ (k, c) | c \in \pi_{\text{add}} \} - \{ (k, c) | c \in \pi_{\text{del}} \} \]

**Definition 3.** The round-robin metafunction implements a scheduling policy, rotating the actors in a configuration:

\[ \text{round-robin} : L^\Sigma \times \text{ActorState} \rightarrow \text{Config} \]

\[ \text{round-robin}(\left[ \cdot \right]; \hat{R}; A_I(\ell \mapsto \square)A_Q), \Sigma) = \left[ \cdot \right]; \hat{R}; A_I \overrightarrow{A_Q}(\ell \mapsto \Sigma) \]

**Properties.** The semantics satisfies several important theorems.
Theorem 4 (Dataspace Soundness [40, theorem 4.17]). Either

- \( \Sigma \stackrel{\rightarrow_{ds}}{\rightarrow} \Sigma_I \); or
- for all \( \Sigma' \), \( \Sigma \stackrel{\rightarrow_{ds}}{\rightarrow} \Sigma' \) implies there exists \( \Sigma'' \) such that \( \Sigma' \stackrel{\rightarrow_{ds}}{\rightarrow} \Sigma'' \)

□

When all individual actors meet the interface, dataspaces enjoy a progress property.

Lemma 5 (Progress of Dataspace Configurations [40, lemma 4.19]). Dataspace configurations are either inert or may further reduce. □

Lemma 6 (Deterministic Evaluation [40, theorem 4.20]). For any \( \Sigma \), there exists at most one \( \Sigma' \) such that \( \Sigma \stackrel{\rightarrow_{ds}}{\rightarrow} \Sigma' \). □

A dataspace applies an actor’s behavior function to only those assertions in which the actor has expressed an interest.

Theorem 7 (Soundness of Routing [40, theorem 4.35]). If \( C_0 \stackrel{\rightarrow_{ds}}{\rightarrow} C_n \) where \( C_n \) is a configuration that is about to dispatch an event to actor \( \ell \):

\[
\begin{array}{c}
\langle \cdot \rangle; R_n ; \overline{A}_1(\ell \mapsto \overline{\pi}'(\pi^+ / \pi^-) \triangleright (f, u) \triangleright \overline{a}_n)\overline{A}_Q \end{array}
\]

then there is some \( C_i, i < n \), with \( C_i = \langle (k, a) ; R_i ; \overline{A} \rangle \) such that the actor has a stated interest in each delivered assertion:

\[
\{ (\ell, \text{observe } c) \mid c \in (\pi^+ \cup \pi^-) \} \subseteq R_i
\]

□

2.3 A STREAMLINED SEMANTICS

The preceding section describes the evolution of dataspace systems as a series of incremental steps. It broke down communication as a series of transitions such as dispatching an event to a single actor; gathering an output action from a single actor; interpreting a single action in a queue; and booting one new actor. This section defines a condensed semantics more suitable for the purposes of reasoning about dataspace communication, that is, how the set of assertions in a dataspace evolves. Chapter 5 in particular has a need for such a semantics.

Figure 2.8 defines the changes to the abstract syntax and the revised semantics of dataspace programs. The queue of pending actions in the dataspace only contains patches. Actor states contain a single pending event, rather than a queue of incoming events. The pending event accumulates a patch for the actor between its turns. 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.

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 queues of pending patches. The update metafunction handles
a single queue item, updating the dataspace’s assertion table and dispatching events
via \(bc_A\). Actors accumulate a resulting pending event comprised of an aggregate patch
and collection of messages, handled by the \(\text{compose}_{\text{pe}}\) function. Next, the \(\text{dispatch}_{\text{DS}}\)
metafunction applies the behavior of each actor to its pending event, producing an
updated actor state and patch, message, and spawn actions. The resulting spawn
actions are boot-ed, yielding a new address, actor state, and assertions. Finally, all of
the patches, messages, and boot-ed assertions are collected into new queues to yield the
next configuration. It includes the updated assertion table and freshly booted actors.

\[
\begin{align*}
\text{DS} &\in \text{Dataspace} = [(\ell, \Delta) ; R ; A] \\
\Sigma &\in \text{ActorState} = \text{pe} \triangleright B \\
\text{pe} &\in \text{PendEvt} = \Delta \\
f &\in \text{BehFun} = \exists St. \text{PendEvt} \times St \rightarrow St \times \text{Action}
\end{align*}
\]

\[
\begin{align*}
[(\ell \act, \Delta) ; R ; A] &\rightarrow_{\text{DS}} [(\ell', \Delta') \cdot (\ell', \pi ; \emptyset) ; R' \cdot (\ell_A \rightarrow \Sigma') \cdot (\ell' \rightarrow \Sigma')]
\end{align*}
\]

where

\[
\begin{align*}
R' &\cdot \ell_A \rightarrow \Sigma \\
\Sigma', \Delta', \pi &\rightarrow \text{foldL update} (\ell \act, \Delta) (R, A) \\
\Sigma' &\rightarrow \text{dispatch}_{\text{DS}} (\Sigma) \\
\pi &\rightarrow \text{boot}_{\Sigma} (\Sigma)
\end{align*}
\]

with \(\ell'\) fresh in \(\ell_A\)

Figure 2.8: Dataspace Syntax and Semantics

**Definition 8 (DSi).** A *run* of a dataspace configuration \(\text{DS}\) is a potentially infinite
sequence of dataspace configurations, where each neighboring pair is related by the
transition relation:

\[
\text{DS} \rightarrow_{\text{DS}} \text{DS}_1 \rightarrow_{\text{DS}} \text{DS}_2 \rightarrow_{\text{DS}} \ldots
\]

It is a run that represents the temporal behavior of a dataspace program and its
individual actors, and it is thus the foundation for analyzing temporal behaviors in
chapter 5.

**Definition 9 (\text{i}n\text{ert}_{\text{DS}}(\text{DS}))**. A dataspace configuration \(\text{DS}\) is inert if its queue of
pending actions is empty: \(\text{DS} = [\cdot ; R ; A]\).

**Theorem 10 (Dataspace Soundness)**. Either \(\text{i}n\text{ert}_{\text{DS}}(\text{DS})\) or there exists \(\text{DS}'\) such that
\(\text{DS} \rightarrow_{\text{DS}} \text{DS}'\).
2.3.1 Metafunctions

The semantics refers to several metafunctions.

**Definition 11 (boot$_\Sigma$).** The boot$_\Sigma$ function boots a process description to an actor state:

$$\text{boot}_\Sigma : P \rightarrow \Sigma \times \pi$$

$$\text{boot}_\Sigma(\text{actor } f \triangleright \pi) = \emptyset / \emptyset \triangleright (f, \pi)$$

**Definition 12 (update).** The update metafunction is the work horse of dataspace event dispatch.

$$\text{update} : (R \times \Delta) \times (\ell \times \text{Evt}) \rightarrow R \times \Delta$$

$$\text{update}((R, \Delta), (\ell, \Delta)) = R \oplus_R (\Delta, \ell), b_\Delta(\ell, R, \Delta, A)$$

**Definition 13 (bc$_\Delta$).** The bc$_\Delta$ metafunction is much the same, but it updates the pending event for an actor rather than updating a queue.

$$b_\Delta : \ell \times R \times \text{Evt} \times A_Q \rightarrow A_Q$$

$$b_\Delta(\ell, R_{old}, \Delta, A) = \begin{cases} \ell \mapsto \text{compose}_{\text{pe}}(\Delta_{fb}) \triangleright B & \text{if } \ell = \ell_{evt} \\ \ell \mapsto \text{compose}_{\text{pe}}(\Delta_{other}) \triangleright B & \text{if } \ell \neq \ell_{evt} \end{cases}$$

where

$$\Delta = \pi_{\text{add}} / \pi_{\text{del}}$$

$$A = \ell \mapsto \text{pe} \triangleright B$$

$$\Delta_{fb} = \{c \mid c \in \pi_{\text{add}}(\text{observe } c, \ell) \in R_{new} \} \cup \{c \mid c \in \pi_{\text{add}} / \pi_{\text{del}}, \text{observe } c \in \pi_{\text{del}}\}$$

$$\Delta_{other} = \{c \mid c \in \pi_{\text{add}}(\text{observe } c, \ell) \in R_{old}\}$$

$$\pi_{\bullet} = \{c \mid (c, \ell') \in R_{old}, \ell' \neq \ell_{evt}\}$$

$$R_{new} = R_{old} \oplus_R (\pi_{\text{add}} / \pi_{\text{del}}, \ell)$$

$$\pi_{\text{add}} = \pi_{\text{add}} / \pi_{\text{del}}$$

$$\pi_{\text{del}} = \{c \mid (c, \ell') \in R_{old}\}$$

$$\pi_{\text{del}}^\bullet = \pi_{\text{del}} / \pi_{\text{add}}$$

The compose$_{\text{pe}}$ helper updates an actor’s pending event.

$$\text{compose}_\text{pe} : \text{pe} \times \text{Evt} \rightarrow \text{pe}$$

$$\text{compose}_\text{pe}(\Delta, \Delta') = \Delta \circ \Delta'$$

The $\circ$ operator describes the effect of applying two patches in sequence, maintaining disjointness.

$$\frac{\pi_1^\circ}{\pi_2} : \Delta \times \Delta \rightarrow \Delta$$

$$\frac{\pi_1^\circ}{\pi_2} = \frac{(\pi_1^\circ - \pi_2^\circ) \cup \pi_2^\circ}{\pi_2^\circ - \pi_2}$$
Definition 14 ($\oplus_R$). The $\oplus_R$ operator updates the table of active assertions based on an actor’s patch.

$$
\oplus_R : R \times (\Delta, \ell) \rightarrow R
$$
$$
R \oplus_R (\pi^+, \pi^-, \ell) = R \cup \{(c, \ell) \mid c \in \pi^+\} - \{(c, \ell) \mid c \in \pi^-\}
$$

Definition 15 ($\text{dispatch}_{DS}$). The $\text{dispatch}_{DS}$ function invokes an actor’s behavior on its pending event, if non-empty.

$$
\text{dispatch}_{DS} : \Sigma \rightarrow \Sigma \times \Delta \times P
$$
$$
\text{dispatch}_{DS}(pe \triangleright (f, v)) =
\begin{cases}
pe \triangleright (f, v), \emptyset / \emptyset, \ & \text{if } pe = \emptyset / \emptyset \\
\emptyset / \emptyset \triangleright (f, v'), \Delta, P \ & \text{otherwise}
\end{cases}
$$

where
$$
v', \vec{a} = f(pe, v)
$$
$$
\Delta = \Delta_0 \circ \Delta_1 \circ \ldots \text{ for } \Delta_i \in \vec{a}
$$
$$
\vec{P} = P_0 \cdot P_1 \ldots \text{ for } P_i \in \vec{a}
$$

2.3.2 A Note on the Relationship Between the Semantics

The streamlined semantics of this section provides actors with a different view of the dataspace’s assertions when compared to the semantics of section 2.2. In this section, when an actor receives an event, the patch describes the aggregate change to the dataspace’s assertions since the actor’s last turn. On the other hand, in section 2.2, each event for an actor stems from one of two sources:

1. the actor itself introducing new interests, in which case the patch describes the existing matches, or
2. a different actor updating its assertions, in which case the patch describes the relevant updates.

Since an actor can keep track of when it introduces new interests, this setup introduces the ability to reason about the provenance of assertions. That is, an actor can deduce that each assertion in a given patch is the product of a single other actor’s turn. This aspect is at odds with the anonymous, loosely coupled nature of dataspace communication.
Designing a type system for dataspaces requires consideration of the semantics of both communication and computation, with special attention to the intersection of the two concerns. This chapter presents a design that meets this high-level criterion as follows:

- The language of types accounts for the flexible nature of dataspace communication with “true” (set) unions [78] for describing sets of assertions. Dataspaces are about commingling actors: each actor partakes in, and each assertion potentially pertains to, several conversations. Union types mirror the overlapping conversations, making them suitable for describing the (sets of) assertions in a dataspace.

- Communication and computation coincide in the functions used to express actor behavior. Dataspace event dispatch (communication) becomes function application (computation). The return values of behavior functions give rise to routing events. These functions warrant additional checking on inputs and outputs, accounting for their dual purpose.

- Additionally, behavior functions must terminate on all inputs. But, writing actors in a language with recursion permits diverging programs. To address this issue, the system employs the standard method of basing the type system on the simply-typed λ-calculus plus an induction schema per recursive type [63], which is known to make for a terminating language.

- Finally, the computational constructs for creating and accessing the sets of assertions used for communication introduce new possibilities for computational errors. Hence, the type system prohibits creating assertion sets with higher-order data or selecting an infinite branch of an assertion set with project.

### 3.1 Types, Informally

This section is an informal introduction to this somewhat unusual combination of ideas.

**Unions.** Union types are the basic building block of typed dataspace actors. In dataspaces, groups of actors participate in conversations where each utterance takes the form of an assertion. Describing these conversations with types means grouping together related assertions. Unions capture the multi-party, overlapping nature of dataspace conversations.¹

¹ Such types do not account for temporal or substructural constraints on exchanges; see section 3.5.
For instance, in the chat room example from section 2.1, actors communicate presence information through in-room assertions and interest in those assertions. Each in-room assertion has type \((\text{InRoomT String String})\). A value of type \((\text{InRoomT String String})\) is an assertion \((\text{in-room } v_1 \; v_2)\) where both \(v_1\) and \(v_2\) are strings. The camel-cased name and final “T” is a convention for such type constructors. The struct form defines the plain \text{InRoom} type name to be an alias for the type constructor fully instantiated with the default type parameters, such as \((\text{InRoomT String String})\).

An assertion of interest in other participants, such as \((\text{observe (in-room } * \; \text{"FBI")})\), has type \((\text{Observe (InRoomT } * \; \text{String)})\). The type uses the parameterization of the \text{InRoomT} constructor to allow wildcard interest in usernames but limit interest to named rooms. As a type, \(\ast\) stands for all possible assertions, including concrete strings, numbers, or interests, as well as any infinite set of assertions arising from \{ \ldots \} (set creation) expressions containing \(\ast\).

The union type \text{PresenceAssertions} describes the “Room Presence” conversation:

\[
\text{(define-type PresenceAssertions (U InRoom (Observe (InRoomT } \ast \; \text{String}))})\)
\]

It includes both the assertion made by a user-agent to signal its presence in a particular room as well as the assertion of interest used to monitor its peers. Together, they describe the possible utterances in the conversation between user-agent actors concerning chat-room presence.

Grouping together the types of each assertion pertaining to a conversation in a union provides useful documentation and aids in separate, modular programming of actors. While a single actor may express only a subset of the assertions, the flexible nature of union types allows the composition of overlapping conversations. For example, the room-list actor monitors in-room assertions and publishes a list of results. The InRoom assertions, the room-list actor’s interest in them,\(^2\) the published RoomList, and the interest used by other actors to learn the list forms an “Available Rooms” conversation:

\[
\text{(define-type RoomAssertions (U InRoom (Observe (InRoomT } \ast \; \ast \; \text{String})) (RoomListT (List String)) (Observe (RoomListT } \ast \; \ast \; \text{String}))})\)
\]

The “Room Presence” and “Available Rooms” conversations overlap, with each in-room assertion playing a role in both. In one conversation an assertion signifies the existence of a particular user, while in the other it signifies the existence of a particular room.

\(^2\) The type of interest in presence employed by the room-list actor, \((\text{Observe (InRoomT } \ast \; \ast \; \text{String}))\), reflects the potential for wildcard interest in all possible rooms, unlike that of the user agent.
As figure 2.3 shows, a user-agent actor performs additional communication to notify the connected user of specific events. This conversation consists of user-notification structures, described by the type

```scheme
(define-type NotificationAssertions
  (U (UserNotificationT String)
      (Observe (UserNotificationT *))))
```

The full implementation includes other conversations for network connectivity, sending chat messages, and changing rooms, each with a similar type: NetworkAssertions, ChatAssertions, RoomAssertions, and so on.

By describing the conversations in isolation, each actor can be implemented in terms of only those conversations in which it participates. The user-agent described here may elide NetworkAssertions, among others:

```scheme
(define-type UserAgentAssertions
  (U PresenceAssertions
         RoomAssertions
         NotificationAssertions))
```

```scheme
(define-type ChatDataspace
  (U PresenceAssertions
       NotificationAssertions
       NetworkAssertions
       ChatAssertions
       RoomAssertions))
```

Figure 3.1: Chat Server Communication Type

**DATASPACE**. Taking the union of all the conversations, as shown in figure 3.1, yields a type that describes the conversations in the entire dataspace. Supplying that type to a typed dataspace constructor,

```scheme
(dataspace ChatDataspace
  (actor ...)
  ...)
```

... demands that all assertions have type ChatDataspace. The annotation is referred to as the communication type of the dataspace. The communication type is an agreement among the actors, both limiting each individual’s actions as well as enabling typed reasoning about the behavior of peers. In the chat dataspace, the type permits the user-agent actor’s presence assertion (`in-room "Mo" "FBI"`), of type InRoom, and interest
(observe (in-room "FBI")), with type (Observe (InRoomT ∗ String)), because of their inclusion in the ChatDataspace type via PresenceAssertions. By contrast, the ChatDataspace type prohibits actors that assert (in-room "Marvin" 42), because the second slot is not a string, or overly-broad queries such as (observe ∗), because ∗ is not a subtype of any of the constructors that appear under observe in ChatDataspace.

While the communication type restricts the assertions an actor may make, it also enables reasoning about the shape of assertions that may match an expressed interest. The key assurance of the ChatDataspace type is that any assertion matching an expressed interest in in-room assertions is an in-room struct where both slots contain strings. That is, in any set of assertions a sent to an actor, the set corresponding to the names of connected users, \{v₁ | (in-room v₁ v₂) ∈ a\}, is a finite set of strings, and similarly for the set of values in the second slot, the names of inhabited rooms. Consequently, the behavior functions of actors in the dataspace may safely use project to analyze the names of individual users and rooms without triggering an error.

**Actors and Simple Behavior Functions.** Every individual actor operates within a dataspace, making and withdrawing assertions, processing assertions, and spawning further actors, each of which also resides in the dataspace. In typed programs, the dataspace contains only assertions belonging to a specific type—the communication type \(\tau_c\). Hence, \(\tau_c\) plays a central role in checking individual actor specifications.

The type for an actor must then reflect the type \(\tau_c\) of the dataspace in which it is going to run. A developer can use this type to rationalize the initial assertions and, most importantly, the code for the behavior function. Specifically, the developer must ensure that the behavior function (1) produces only actions—assertions as well as spawned actors—that the dataspace's type permits; (2) is prepared to deal with any event to which the dataspace may apply it; and (3) spawns only actors that recursively obey these constraints, too.

A translation of these insights calls for equipping an actor term

\[(\text{actor } \text{behavior state assertions})\]

with an interface type. This step resembles the addition of a parameter type to \(\lambda\)-terms during the design of a simple type system for languages based on the \(\lambda\)-calculus. Since an actor communicates within a dataspace of some communication type, the best way to signal this assumption is with an annotation \(\tau_c\) that requests this match:

\[(\text{actor } \tau_c \text{ behavior state assertions}).\]

In contrast to function application, actor application is implicit. An actor term is used by submitting it to the dataspace as an action. In dispatching events, the dataspace applies the actor's behavior function and interprets the resulting actions. These semantics—dispatching events and interpreting actions—are not represented in
the surface syntax of the program, meaning it is not possible to check the use of an
actor independently from its use in a dataspace. Assigning the type

\[(\text{Actor } \tau_c)\]

to an actor term expresses this insight. By implication, it becomes straightforward to
check a dataspace term. All initial actors must have the type \(\text{(Actor } \tau_c)\) if \(\tau_c\) is the
type of the dataspace.

Validating that an actor has a given type proceeds according to the three steps above:

1. The assertions of an actor come from two sources: either as part of its initial
assertions or as an action produced by its behavior function. The former requires
checking that the type of the initial assertion set is included by the assumed
communication type \(\tau_c\). For the latter, recall the informal signature of behavior
functions from section 2.1.1:

\[\text{Event} \times \text{State} \rightarrow \text{State} \times \text{Instructions}\]

For the moment, let us simplify things even further by considering the behavior
function as taking in assertions of some type \(\tau_{in}\) and outputting assertions of
another type, \(\tau_{out}\):

\[\tau_{in} \rightarrow \tau_{out}\]

Conceptually, \(\tau_{in}\) and \(\tau_{out}\) are the essence of the \text{Event} and \text{Instructions} types,
respectively. Validating the assertions stated by the actor then entails checking
that \(\tau_c\) includes each type of assertion from \(\tau_{out}\).

2. Dataspaces compute and route events according to expressed interests. Hence,
the assertions constructed with \text{Observe} in \(\tau_{out}\) define a bound on possible events.
Type checking uses this bound together with the rest of the assertions in the
dataspace, represented by \(\tau_c\), to predict the types of events produced by routing
when the program runs. To make this prediction, the type checker takes the
intersection between assertions of interest in \(\tau_{out}\) with \(\tau_c\). The result is a type
describing all potential events the behavior function may be applied to, which
must be a subtype of \(\tau_{in}\).

Concretely, in the case of the user-agent actor \(\tau_{out}\) is

\[\text{(U (Observe (InRoomT \star String)))}\]
\[\text{InRoom}\]
\[\text{UserNotification}\]

According to the type, the actor might assert \((\text{observe (in-room } \star \text{"FBI")})\). It
would then receive a notification containing all in-room assertions with the string
"FBI" in the dataspace. Inspecting the ChatDataspace type of figure 3.1, the type
of potentially overlapping assertions is InRoom. Consequently, the user-agent
actor’s behavior function input type \(\tau_{in}\) must accommodate such assertions.
3. Finally, an actor may spawn other actors into its dataspace. If every one of these actors has type \((\text{Actor } \tau_c)\), they all obey the communication discipline of the surrounding dataspace.

**Manipulating Assertion Sets.** The type system prevents the creation of illegal assertion sets by stratifying types into two levels. The first level, flat types, corresponds to the plain data suitable for sharing in the dataspace: \(\text{Int}, \text{String}\), and so on, as well as type constructors such as lists, structs, and unions when applied to other flat types. The second level is everything else—values that cannot easily be compared for equality, such as functions, objects, and actors. Typing a set-creation form \(\{e \ldots\}\) then checks that each element \(e\) has a flat type, i.e., *not* a function, object, or actor.

The type \((\text{AssertionSet } \tau)\) describes a set of assertions of type \(\tau\) arising from an expression \(\{e \ldots\}\). A patch \((\text{patch } e e)\) is essentially a pair of assertion sets, thus the type \((\text{Patch } \tau^+ \tau^-)\) records the type of assertions to add, \(\tau^+_+\), and the type of assertions to withdraw, \(\tau^-\).

The abbreviation \((\text{Event } \tau)\) stands for \((\text{Patch } \tau \tau)\), signifying that dataspaces notify actors of both the appearance and disappearance of assertions matching an interest. An actor may request the retraction of an assertion it is not presently making, which is a no-op. An actor may make use of this fact by issuing an overly-broad retraction, alleviating some responsibility for tracking currently made assertions. For example, an actor may submit \((\text{patch } \emptyset \{\star\})\) to withdraw all of its current assertions. To account for this fact, the abbreviation

\[
(\text{Action } \tau \sigma) \overset{df}{=} (\text{U} (\text{Patch } \tau \star) (\text{Actor } \sigma))
\]

describes actor actions. The type allows any set of assertions to be withdrawn, as well as the potential to require spawned actors to operate at the communication type \(\sigma\).

Typed assertion-set projection differs from the untyped version in several ways. The first is a syntactic change to patterns. In order to facilitate type checking, pattern variables come with a type annotation, as in \$name: \text{String}\).

The type of this pattern variable is then \((\text{Bind } \text{String})\), while the \_ pattern has type \(\text{Discard}\). Type checking additionally verifies that the expressions within patterns are well-typed.

Detecting erroneous uses of \text{project} involves considering paths to binding patterns and \(\star\) in potentially matching types. Recall that \text{project} raises an error when a binding pattern, such as \$name: \text{String}, has an infinite number of matches in the given set, as in the following expression:

\[
(\text{project } \{(\text{in-room } \$name: \text{String } "\text{FBI}" ) \{(\text{in-room } \star \star)\}})
\]

\[\text{name}\]

3 The implementation can infer these annotations, but the examples here include them to distinguish typed and untyped code.
Conceptually, performing the operation requires iterating over the infinite set
\[
\{ v \mid (\text{in-room } v \ "FBI") \in \{ (\text{in-room } \star \star) \}\}
\]

The cause of the error is not simply because the given set is conceptually infinite. Often, the structure of the pattern provides enough information to discriminate most elements of the set. In the expression
\[
\text{(project \([(\text{in-room } \$name:\text{String } "FBI") \{(\text{in-room } "Mo" \star)\}]\ name)}
\]
the matched set,
\[
\{ v \mid (\text{in-room } v \ "FBI") \in \{ (\text{in-room } "Mo" \star)\}\} = \{ "Mo" \},
\]
is finite, thus evaluation poses no issues. An error occurs exactly when a binding variable in the pattern corresponds to a wildcard \( \star \) in the assertion set. The type system therefore tracks both uses—the latter by assigning \( \star \) the type \( \star \)—and analyzes the type of the pattern against the type of the contents of the set. The potential for the pattern to match assertions is determined by computing the overlapping elements of the types. A type error arises only if there is a common path through both types that may potentially match, leading to \( \text{(Bind } \tau) \) in the pattern and \( \star \) in the set.

**Termination.** In order for dataspace programs to make progress, individual actors must terminate—either normally or exceptionally—in response to every event. This assumption is easily violated when programming with general-purpose constructs such as functions. The type system disallows recursive functions, but still allows for using recursive data structures such as lists via inductive schemas. Recursive data structures must be used via an inductive eliminator in the shape of a folding loop.

**Actor Subtyping.** Subtyping for actor actions aids modular development of actors and permits type-level constraints to be imposed on individual actors, rather than the entire dataspace. For example, the user-agent actor can be developed using a communication type that describes only those conversations in which it participates:

\[
\text{(actor UserAgentAssertions ...)}
\]
yielding a term of type \( \text{(Actor UserAgentAssertions)} \). Ultimately, however, user-agent actors operate in a dataspace with communication type \( \text{ChatDataspace} \). Hence, the system must ensure that the difference between the two communication types does not invalidate the reasoning by which type checking the user-agent actor first succeeded.

Checking the actor creation action computes an intersection between the assertions of interest made by the user-agent actor with \( \text{UserAgentAssertions} \) to determine the events it might receive. In a dataspace with a different type, such as \( \text{ChatDataspace} \),
the actor’s interests might match different types of assertions, which may potentially be incompatible with its behavior function type. The question, then, is whether the actor’s interests lead to “surprising” events when operating in a different type of dataspace.

Actor subtyping answers this question. The first item to check is that all of the user-agent’s assertions are allowed in the greater chat dataspace, that is, UserAgentAssertions must be a subtype of ChatDataspace. Next, we must make sure that running in a ChatDataspace context does not yield surprising events, i.e., events not considered the first time we checked the user-agent. The UserAgentAssertions type permits one type of interest,

\[ \tau_{\text{obs}} = (\text{Observe (InRoomT}) \star \star) \]

Intersecting \( \tau_{\text{obs}} \) with ChatDataspace yields the possible type of events the actor receives when run in the dataspace, \((\text{InRoomT String String})\).

Since \((\text{InRoomT String String})\) is in UserAgentAssertions, it is also in the intersection of \( \tau_{\text{obs}} \) and UserAgentAssertions. Consequently, \((\text{InRoomT String String})\) events have already been considered, and deemed safe, against the input type of the user-agent actor’s behavior function during the checking of the corresponding actor form.

### 3.1.1 Typing the Chat Room

Figure 3.2 displays the typed version of the code from figure 2.3, highlighting the changes. The primary difference between the two figures is the addition of the UserAgentAssertions and ChatDataspace type abbreviations (figure 3.1). Otherwise, there are minor changes to insert type annotations on function parameters, binding patterns, and actor-spawning expressions. The example also makes use of the abbreviation \( \perp \) for the empty union.

The behavior function comes with a narrow type of input event,

\((\text{Event InRoom})\)

containing only one of the many forms of assertions with which actors communicate in the dataspace. Because the actor states only one type of interest, the type system is able to verify that incoming events do indeed have such a refined type. The example illustrates how a highly expressive type system can easily validate complex confluences of communication and computation.

The typed chat dataspace above rules out simple mistakes such as using a number instead of a string for a username. These mistakes can result in non-local actor failure: faults in actors that consume, rather than produce, bad presence information. The ChatDataspace communication type prevents such an actor from being introduced into the typed dataspace by specifying that room names are only ever finite sets of strings.
(define-type UserAgentAssertions ...
... defined on page 25 ...)

(define-type ChatDataspace ...
... defined in figure 3.1 ...)

(define (main)
dataspace ChatDataspace
(list
(create-user-agent "Mo" "FBI"
(create-room-list ...)
...)))

(define (create-user-agent [name : String]
  [initial-room : String]
  ➔ (Actor UserAgentAssertions))

(actor UserAgentAssertions
  ;; behavior: notify the user as peers enter and
  ;; leave the current room.
  (lambda ([event : (Event InRoom)]
    [current-room : String])
    (define arrivals
      (announce (patch-added event) current-room " arrived."))
    (define departures
      (announce (patch-removed event) current-room " departed."))
    (transition current-room
      (append arrivals departures)))
  initial-room
  {{(in-room name initial-room)
    (observe (in-room ⋆ initial-room))}})

;; Create user-notification patches based on
;; presence assertions in an event.
(define (announce [assertions : (AssertionSet InRoom)]
  [current-room : String]
  [message : String]
  ➔ (List (Patch UserNotification ⊥))
  (project [{(in-room $who: String current-room) assertions]
    (patch {{(user-notification (string-append who message))} {}})))

Figure 3.2: The typed user agent actor

**Constraining the User-Agent.** The typed chat dataspace permits queries over both users in a specific room, (Observe (InRoomT ⋆ String)) and over all rooms, (Observe (InRoomT ⋆ *)). The former are used by user-agent actors to monitor the
presence of peers in a room while the latter are used to aggregate information about which rooms exist. Since UserAgentAssertions includes both PresenceAssertions and RoomAssertions, the type permits user-agent actors to express both interest in specific room names and wildcard interest in every room. As discussed in section 2.1.4, the developer may wish to enforce that user-agents do not make overly broad queries. For example, private channel names may leak to an actor that asserts \((\text{observe } (\text{in-room } \star \star))\). To avoid such leakage, the developer may ascribe a refined communication type to user-agent actors:

\[
\text{(define-type RestrictedUserAgent }
\text{(U InRoom }
\text{(Observe (InRoomT } \star \text{ String)})
\text{ NotificationAssertions))}
\]

This restrictive type enforces that queries are only over specific room names. Actor subtyping permits using the refined user-agent actor in the chat server dataspace.

3.1.2 Revisiting the Arithmetic Service

The arithmetic service dataspace from section 2.1.3 suffered from an error that arises from \(\star\) assertions. Types prevent the scenario where sum assertions are abused to request difference calculations. The program may use the communication type

\[
\text{(define-type ArithmeticAssertions }
\text{(U (SumT Int Int Int) }
\text{(Observe (SumT Int Int \star))}
\text{ (Observe (Observe (SumT \star \star \star)))))}
\]

and this type rules out difference-calculation requests.

Such a difference request assertion, say \((\text{observe } (\text{sum } 4 \star 9))\), would have type \((\text{Observe } (\text{SumT Int } \star \text{ Int}))\), which is not subsumed by ArithmeticAssertions. In particular, because \(\star\) stands for all types of assertions, including strings and structures, it is incompatible with the occurrence of Int in the corresponding position in ArithmeticAssertions. An actor with such an output fails to type check in the ArithmeticAssertions dataspace, thus preventing the dynamic error mentioned in section 2.1.4.

An alternative communication type could be more permissive with regard to interest in sum assertions:

\[
\text{(define-type PermissiveInterests }
\text{(U (SumT Int Int Int) }
\text{(Observe (SumT \star \star \star))}
\text{ (Observe (Observe (SumT \star \star \star)))))}
\]
A dataspace of this type allows difference requests. However, now a type error arises for the actor implementing the arithmetic service. Recall that the service iterates over incoming assertion sets \( e \),

\[
\text{project } \left\{ \begin{array}{l}
\text{(observe (observe (sum $x$:Int $y$:Int _)) e)} \\
\text{(sum x y (+ x y))}
\end{array} \right. 
\]

This projection occurs inside the body of a behavior function, which assumes some type for the incoming assertions contained by \( e \). If that \( e \) has type \text{PermissiveInterests}, the type system signals an error, since the path to \$x$:Int in the pattern leads to \( \star \) in the set. By contrast, assuming the contents of the set \( e \) have type \text{ArithmeticAssertions} allows the projection to type check.

The type system detects a mismatch between the assumption and reality when such an actor action is submitted to a dataspace with communication type \text{PermissiveInterests}. Concretely, the actor makes an assertion of interest \( \text{(observe (observe (sum } \star \star \star))} \) to learn about sum requests, with type

\[
\text{(Observe (Observe (SumT } \star \star \star))}
\]

Potentially matching assertions are in the intersection with type \text{PermissiveInterests},

\[
\text{(Observe (SumT } \star \star \star))
\]

which is \text{not} a subtype of \text{ArithmeticAssertions}. Finally, checking the actor action succeeds if the supplied communication type annotation is \text{ArithmeticAssertions}. Actor subtyping prevents instantiating such an actor in the \text{PermissiveInterests} dataspace through a similar failed type-checking attempt.

The arithmetic service actor typifies the interplay between projection, function, and actor typing. When writing a behavior function, the developer analyzes incoming assertion sets using \text{project}. In order for the function itself to type check, the set must have a type compatible with the supplied patterns. These constraints flow outward, to the function’s parameter for incoming events, and they then become part of the domain of the function’s type. When the function is used as the behavior component of an actor action, the type system finds the assumptions in the domain of the function type and compares them with the reality of the surrounding dataspace. Only when all of these elements agree are programs well-typed.

### 3.2 Actors, Types, Formally

Figure 3.3 introduces \( \lambda_{ds} \), a model language for articulating untyped dataspace actors. It is representative of ECMAScript and Racket prototypes, i.e., functional languages that extend the \( \lambda \)-calculus syntax with means to interface with dataspaces.

A complete program in \( \lambda_{ds} \) is a description of a dataspace, dataspace \( M \). The expression \( M \) computes a list of actors to launch at the start of the program. Over
the course of execution, additional actors may be dynamically spawned through actor actions as well as removed due to failure or termination. Section 3.3 describes how this initial description yields a running actor system.

Extensions to the base functional model compute values that, when sent to the surrounding dataspace, trigger certain actions: an assertion of interest (observe $M$); an assertion of fact ($m(\overrightarrow{M})$); the spawning of a new actor (actor $M_b M_s M_a$). The three parts of actor correspond to the behavior function ($M_b$), private state ($M_s$), and initial assertions ($M_a$) of an actor. Additionally, the extensions also include expression forms to create sets of assertions ($\{\overrightarrow{SK}\}$) and compose patches ($M^+/M^-$).

Behavior functions in $\lambda_{ds}$ return a pair of a new state value and a list of actions, as opposed to the quit and transition records of the prototype described in section 2.1.1. The capabilities of the two interfaces are the same; rather than issue an explicit quit,
actors in $\lambda_{ds}$ signal termination by raising an error, triggering their removal from the dataspace.

A set constructor SK describes assertions, ranging from singletons to infinite sets ($\star$); \{ SK \} translates to an assertion set $\pi$ (from figure 2.4) for placement in the dataspace. When an SK expression yields a value not suitable for dataspace assertions, such as a function, the reduction ends in an error. The different variants of assertions employed in a dataspace program correspond to a set of message constructors $m$, which otherwise behave like tuples.

The project form is the key mechanism for de-structuring incoming assertion sets in a behavior function. Specifically,

$$\text{project } \pi \text{ with } P \text{ in } M$$

instantiates $M$ with the bindings of $P$ for each matching assertion in $\pi$ and assembles the results into a list. A pattern’s bindings may match an infinite number of values, as in

$$\text{project } \{ \star \} \text{ with } $x \text{ in } x$$

The semantics interprets such expressions as errors.

The Reduction Semantics of $\lambda_{ds}$

The reduction semantics of $\lambda_{ds}$ is mostly conventional [32]; see figure 3.4. To minimize bookkeeping, the grammar reuses the syntax of assertions from section 2.2. The addition of assertions $c$ and assertion sets $\pi$ requires only a small extension over the standard call-by-value semantics. Assertion sets are created from set constructors with the metafunction make-set. The project metafunction implements projection, which eliminates assertion sets. Section 3.2.1 provides the full definition of each metafunction.

The semantics distinguishes among three sources of errors in order to characterize the soundness of the model precisely:

1. $\text{error}_{\text{prim}}$ arises from application of partial primitive operations;
2. $\text{error}_{\text{h-o}}$ arises from assertion sets containing functions or actors;
3. $\text{error}_{\text{inf}}$ arises when project selects an infinite set of assertions.

3.2.1 $\lambda_{ds}$ Reduction Metafunctions

The reduction semantics of $\lambda_{ds}$ (figure 3.4) relies on several metafunctions that create and analyze assertion sets. This section collects their formal definitions.
Evaluation Syntax

\[ M = \cdots | \pi \quad c \in \text{Assertion} = \text{(defined in fig. 2.4)} \]
\[ \pi \in \text{ASet} = \text{(defined in fig. 2.4)} \]

\[ v \in \text{Val} = \lambda x. M \mid b \]
\[ (\vec{v}) \quad \text{cons } v \quad \text{observe } v \mid m(\vec{v}) \]
\[ \pi \quad \text{actor } v \quad \text{project } \pi \text{ with } v \text{ in } M \]
\[ \star \quad $x \quad \_ \]

\[ E \in \text{Ctx} = \Box \mid E \mid v \mid \overrightarrow{v}E \overrightarrow{M} \mid (\overrightarrow{v}E\overrightarrow{M}) \]
\[ \text{cons } E \mid \text{cons } v \mid \text{observe } E \mid \text{actor } E \mid \text{actor } v \mid \text{actor } v \mid \text{actor } v \]
\[ \star \mid \$x \mid \_ \]

Evaluation

\[ \text{eval}_M (M) = \begin{cases} v \text{ if } M \rightarrow^* v \\ \text{error}_\eta \text{ if } M \rightarrow^* \text{error}_\eta \end{cases} \]

Notions of Reduction

\[ E[(\lambda x. M)v] \rightarrow E[M[x \mapsto v]] \quad \text{(\beta_v)} \]
\[ E[p \overrightarrow{v}] \rightarrow E[v'] \quad \text{where } \delta (p, \overrightarrow{v}) = v' \quad \text{(\delta)} \]
\[ E[p \overrightarrow{v}] \rightarrow \text{error}_{\text{prim}} \quad \text{where } \delta (p, \overrightarrow{v}) \text{ is undefined (\delta-error)} \]
\[ E[\overrightarrow{v}] \rightarrow E[\pi] \text{ where } \text{make-set}(\overrightarrow{v}) = \text{\{make-set\}} \]
\[ E[\overrightarrow{v}] \rightarrow \text{error}_{\text{h-o}} \text{ where } \text{make-set}(\overrightarrow{v}) \text{ is undefined (\text{make-set-error})} \]
\[ E[\text{project } \pi \text{ with } v \text{ in } M] \rightarrow E[M'] \]
\[ E[\text{project } \pi \text{ with } v \text{ in } M] \rightarrow \text{error}_{\text{inf}} \text{ where } \text{project}(\pi, v, M) = M' \text{ (project-error)} \]

Metafunctions

\[ \text{make-set} : \text{Val} \xrightarrow{\text{partial}} \text{ASet} \]
\[ \text{make-set}(\overrightarrow{v}) \text{ translates a vector of values into a dataspace representation}\]
\[ \text{undefined if given higher-order values} \]

\[ \text{project} : \text{ASet} \times \text{Val} \times \text{Expr} \xrightarrow{\text{partial}} \text{Expr} \]
\[ \text{project}(\pi, v_p, M) \text{ creates a list by replacing pattern variables from } v_p \text{, an evaluated}\]
\[ \text{pattern, in } M \text{ with values from matching assertions in } \pi \]
\[ \text{undefined if there is an infinite number of different matches} \]

\[ \delta : \text{PrimOp} \times \text{Val} \xrightarrow{\text{partial}} \text{Val} \]
\[ \text{applies a primitive; undefined in cases due to partial primitives} \]

Figure 3.4: The Essence of the Formal Semantics of \(\lambda_{ds}\)
**Definition 16.** The *make-set* metafunction creates an assertion set from a vector of values:

\[
\text{make-set} : \overrightarrow{\text{Val}} \xrightarrow{\text{partial}} \overrightarrow{\text{ASet}}
\]

\[
\text{make-set}(\overrightarrow{v}) = \bigcup \overrightarrow{\pi} \quad \text{where} \quad \overrightarrow{\pi} = \text{interp}(\overrightarrow{v})
\]

**Definition 17.** The *interp* function maps \(\lambda ds\) values to sets of assertions:

\[
\text{interp} : \overrightarrow{\text{Val}} \xrightarrow{\text{partial}} \overrightarrow{\text{ASet}}
\]

\[
\text{interp}(\star) = \text{Assertion}
\]

\[
\text{interp}(b) = \{b\}
\]

\[
\text{interp}(m(\overrightarrow{v})) = \{m(\overrightarrow{v'}) | (\overrightarrow{v'}) \in \text{interp}(\overrightarrow{v})\}
\]

\[
\text{interp}(()) = \{()\}
\]

\[
\text{interp}((v, \overrightarrow{v})) = \{(x, \overrightarrow{y}) | x \in \text{interp}(v), (\overrightarrow{y'}) \in \text{interp}(\overrightarrow{v})\}
\]

\[
\text{interp}(\text{cons} v_1 v_2) = \{\text{cons} x y | x \in \text{interp}(v_1), y \in \text{interp}(v_2)\}
\]

\[
\text{interp}((\text{observe} v)) = \{\text{observe} x | x \in \text{interp}(v)\}
\]

**Definition 18.** The *project* function analyzes assertion sets with a pattern:

\[
\text{project} : \overrightarrow{\text{ASet}} \times \overrightarrow{\text{Val}} \times \overrightarrow{\text{Expr}} \xrightarrow{\text{partial}} \overrightarrow{\text{Expr}}
\]

\[
\text{project}(\pi, v_p, M) = \text{unroll}(m) \quad \text{if} \quad m \text{ is finite}
\]

\[
\text{where} \quad m = \{\gamma(M) | v \in \pi, \text{match}(v, v_p) = \gamma\}
\]

Successful pattern matches yield substitutions:

Substitutions \(\gamma \in \text{Sub} = \overrightarrow{\text{Var}} \xrightarrow{\text{partial}} \overrightarrow{\text{Val}}\)

where composition \(\gamma_1 \circ \gamma_2\) is defined in the usual manner.

Pattern matching is defined in straightforward fashion:

\[
\text{match} : \overrightarrow{\text{Val}} \times \overrightarrow{\text{Val}} \xrightarrow{\text{partial}} \overrightarrow{\text{Sub}}
\]

\[
\text{match}(v, sx : \tau) = \{(x, v)\}
\]

\[
\text{match}(v, _) = \{\}
\]

\[
\text{match}(b, b) = \{\}
\]

\[
\text{match}(\text{observe} v, \text{observe} v_p) = \text{match}(v, v_p)
\]

\[
\text{match}(m(\overrightarrow{v_n}), m(\overrightarrow{v_{pn}})) = \text{match}(\overrightarrow{v_n}, \overrightarrow{v_{pn}})
\]

\[
\text{match}(\overrightarrow{v_n}, \overrightarrow{v_{pn}}) = \text{match}(v, v_p) \circ \gamma
\]

\[
\text{where} \quad \text{match}(\overrightarrow{v_n}, \overrightarrow{v_{pn}}) = \gamma
\]

\[
\text{match}(\text{cons} v_1 v_2, \text{cons} v_{p1} v_{p2}) = \text{match}(v_2, v_{p2}) \circ \gamma
\]

\[
\text{where} \quad \text{match}(v_1, v_{p1}) = \gamma
\]
\begin{align*}
I &= \text{dataspace } \tau, M \\
\tau, \sigma &\in \text{Type} = \tau \rightarrow \tau \mid B \\
M &= \ldots \\
| \lambda x : \tau. M \\
| \text{fold } M M M \\
| \text{actor } \tau, M_0, M_a \\
P &= \ldots \\
| \$x : \tau \\
B &\in \text{BaseTy} = \text{base types: String, Int, etc.} \\
v &= \ldots \\
| \lambda x : \tau. M \\
| \text{actor } \tau, v_0, v_a \\
| \text{dataspace } \tau, v \\
| \$x : \tau \\
\Gamma &\in \text{Env} = \bar{x : \tau} \\
\bot &\overset{df}{=} \cup \\
\text{Action } \tau \sigma &\overset{df}{=} (\text{Patch } \tau \ast) \cup (\text{Actor } \sigma)
\end{align*}

Figure 3.5: Typed Syntax of $\lambda_{ds}$

The \textit{unroll} function translates the set of results to a list expression:

\[
\begin{align*}
\text{unroll} & : \mathcal{P}(\text{Expr}) \xrightarrow{\text{partial}} \text{Expr} \\
\text{unroll}(\emptyset) &= \text{nil} \\
\text{unroll}(\{M\} \cup S) &= \text{cons } M \text{ unroll}(S)
\end{align*}
\]

While the unrolling operation does not specify the order of elements, assuming a fixed ordering for the selection of elements to form the given set suffices to make the definition deterministic.

### 3.2.2 Types for Dataspaces and Actors

Figure 3.5 extends the syntax of $\lambda_{ds}$ with simple types, giving rise to $\lambda_{ds}^{\cup}$. The primary difference is the presence of type annotations in functions, actors, dataspaces, and patterns; additionally, expressions now include \text{fold } M_c M_u M_l, a representative induction scheme for iterating over lists. Reduction of \text{fold} terms is standard; lists unfold to applications of $M_c$ with base case $M_n$.

The language of types reflects the underlying expression language and adds union types for dealing with dataspaces communication. A union of types is written $\cup \tau$. When convenient, examples use the infix notation $\tau \cup \tau$. The semantics does not provide any elimination forms for union types. Rather, programs use unions primarily for describing the contents of assertion sets and rely on the pattern supplied to project to
discriminate the branches of a union. The definition of the \textit{project} metafunction is the same as the one in the untyped model, relying on the structure of the pattern to find matching assertions. Therefore, the type of the pattern must predict how matching will proceed during evaluation. For example, let $M$ describe a set of assertions $\tau$, where

$$\tau = m_1(\text{String}) \cup m_2(\text{Int}) \cup m_3()$$

Using a pattern that describes only one of the message constructors, say $m_2$, as in

\texttt{project \(M\) with \(m_2(x : \text{Int})\) in \(x\)}

focuses type checking on only those members of the union that might match the pattern, allowing the projection to ignore $m_1$ and $m_3$. In a full implementation, the addition of occurrence typing \cite{100} should facilitate programming with unions.

The type $\text{Actor} \ \tau_c$ is that of actors in a dataspace with communication type $\tau_c$.

The data constructor $\text{observe}$ is directly reflected as the type constructor $\text{Observe}$. Similarly, $m(\overline{\tau})$ is the type of a message constructed by $m$ with fields $\overline{\tau}$. The type $*$ is the type of assertions created with $\ast$. Thus, the expression \{ $\text{observe}$ (in-room($\ast$ , "FBI")) \} has type

\texttt{AssertionSet (Observe (in-room($\ast$ , String)))}

The type $\text{Patch} \ \tau^+ \ \tau^-$ describes patches $M^+ / M^-$ where $M^+$ has type $\text{AssertionSet} \ \tau^+$ and $M^-$ type $\text{AssertionSet} \ \tau^-$. Finally, $\$ : \tau$ is the type of patterns $\$x : \tau$, \textit{Discard} is the type of $\_\$, and $\text{Action} \ \tau \ \sigma$ abbreviates $(\text{Patch} \ \tau \ \ast) \cup (\text{Actor} \ \sigma)$.

3.2.3 \textit{Static Semantics}

The purpose of the typing rules for $\lambda_{\cup ds}$ in figure 3.6 is to establish three key invariants using a standard judgment. First, actors place only valid data in their dataspace, i.e., first-order sets of assertions at the specified communication type. Second, actors extract only finite subsets of assertions via $\text{project}$. Third, well-typed actors are terminating; that is, they either signal an error or terminate with a list of actions and a new private state value. This last invariant discharges an assumption of the universal soundness theorem of Garnock-Jones et al \cite{41}.

The typing rule for dataspace programs, $T$-\textit{DATASPACE}, ensures that the communication type $\tau_c$ describes assertions with the premise $\text{flat} (\tau_c)$. The $\text{flat}$ judgment identifies the types of basic, first-order data such as numbers, strings, tuples of numbers, and so on suitable for dataspace assertions. The rule also checks that $M_{\text{boot}}$, the boot actors, all safely operate with communication type $\tau_c$.

\footnote{Unlike the camel-casing convention in the implementation, in the formalism both types and values use the same constructor name, here \textit{in-room}.}
Figure 3.6: Selected Elements of an Actor Type System
According to T-Actor, an actor of shape \( \tau \) \( M_b \) \( M_s \) \( M_a \) may participate in a dataspace of type \( \tau \) when the type of its behavior function, \( M_b \), fits the template

\[
(\text{Patch } \tau_{\text{in}}, \tau_{\text{state}}) \rightarrow (\text{List } (\text{Action } \tau_{\text{out}} \tau_c), \tau_{\text{state}})
\]

and satisfies certain conditions concerning \( \tau_{\text{in}} \), \( \tau_{\text{out}} \), and \( \tau_c \). Specifically, all assertions produced by the behavior function, \( \tau_{\text{out}} \), must be valid utterances in \( \tau_c \). Next, \( \tau_{\text{in}} \), the type of assertions the actor is prepared to handle, must account for the actor’s interests. Intuitively, an actor must be prepared to receive all of the assertions it asks for. The type \( \tau_{\text{in}} \) describes the actor’s assumptions about the sets of assertions it receives, including which subsets may be infinite. The actual assertions it receives arise from dataspace routing, which matches stated interests against all current assertions. The predict-routing metafunction, defined in figure 3.7, mirrors run-time routing in order to predict the types of assertions received by an actor. It computes the overlap between the types of interests expressed by an actor and the types of possible assertions in the dataspace. The assertions the actor may express interest in are exactly those prefixed by \text{Observe} in \( \tau_{\text{in}} \); the metafunction \text{strip-obs} finds all such types, while the \( \cap \) metafunction determines the type representation of the overlap between such interests and the potential assertions in the dataspace, \( \tau_c \) (figure 3.7). Since \( \ast \) stands for all possible assertions, including \text{observe}-prefixed ones, \text{strip-obs} treats \( \ast \) as if it were the unfolding of \text{Observe} \( \ast \). Finally, the second parameter of \( \text{Action } \tau_{\text{out}} \tau_c \) in T-Actor requires the type of any spawned actor to conform to communication type \( \tau_c \).

Rule T-PROJECT eliminates two potential problems from matching a pattern with type \( \tau_p \) against a set of type \( \text{AssertionSet } \tau_s \). First, binding variables in the pattern may have an infinite number of matches. Second, matching assertions may be incompatible with the type associated with binding variables, \( \$: \tau \). The \text{project-safe}(\tau_s, \tau_p) \) judgment enforces this constraint by finding the portion of the analyzed assertion set that may match and bind variables in the pattern. Both aspects are checked by rule PS-CAPTURE (definition 25), which corresponds to an identified match. The \text{finite}(\tau) \) premise ensures matched assertion types do not contain \( \ast \), while \( \tau <: \sigma \) checks that they meet the pattern’s expectations.

The T-\( \pi \) rule describes assertion sets \( \pi \), which are not a part of the surface syntax of \( \lambda^\pi_{ds} \). They arise through the evaluation of \( \{ \text{SK} \} \) forms and via dataspace event dispatch. Employing a standard progress-and-preservation proof technique requires assigning them types. The \( \pi \models \tau \) judgment (definition 26) checks that the set \( \pi \) corresponds to the structure of type \( \tau \) in a way that allows it to be used by \( \lambda^\pi_{ds} \) actors.

For example, consider the type for interest in presence of a chat room:

\[
\tau_q = \text{Observe } (\text{in-room}(\ast, \text{String}))
\]

A set \( \pi \) has type \( \text{AssertionSet } \tau_q \) under these conditions:

- Every element of \( \pi \) must be a message constructed with \text{in-room} holding two fields, prefixed by the \text{observe} constructor.
• The set of values appearing in the second slot of each message must be a finite set of strings. That is,

\[ \{ v_2 \mid \text{observe in-room}(v_1, v_2) \in \pi \} \]

is a finite set of Strings.

• The set of values appearing in the first slot of each message may be any set of assertions, including infinite sets.

The subtyping judgment \( \text{Actor } \tau \triangleleft \text{Actor } \sigma \) ensures that every utterance in \( \tau \) is a valid utterance in \( \sigma \). Furthermore, the judgment must also check for the possibility of interference. That is, transplanting the \( \tau \)-typed actor’s subscriptions to the new \( \sigma \)-typed dataspace must not result in assertions that the actor is unprepared to handle. As in T-Actor, this condition is checked by computing \( \text{predict-routing}(\tau, \sigma) \), a type that reflects routing in dataspaces.

\( \lambda^{|} ds \) Complete Definitions

The following definitions complete the type judgment for \( \lambda^{|} ds \) terms.

**Definition 19.** The \( B \) and \( \Delta \) metafunctions assign sound types to primitive values and operations:

\[
B : \text{BasicVal} \rightarrow \text{BaseTy} \\
\Delta : \text{PrimOp} \times \tau \xrightarrow{\text{partial}} \tau
\]

**Definition 20.** The \( \text{bindings} \) function extracts the type annotations from a pattern; repeated occurrences of the same identifier result in shadowing:

\[
\begin{align*}
\text{bindings} : \text{Pat} & \rightarrow \text{Env} \\
\text{bindings}(\$x : \tau) &= x : \tau \\
\text{bindings}(\langle m \rangle \overrightarrow{P}) &= \text{bindings}(\langle \overrightarrow{P} \rangle) \\
\text{bindings}(\langle P, \overrightarrow{P_n} \rangle) &= \text{bindings}(P_1), \text{bindings}(\langle \overrightarrow{P_n} \rangle) \\
\text{bindings}(\text{observe } P) &= \text{bindings}(P) \\
\text{bindings}(\text{inbound } P) &= \text{bindings}(P) \\
\text{bindings}(\text{outbound } P) &= \text{bindings}(P) \\
\text{bindings}(_{-}) &= \cdot \quad \text{otherwise}
\end{align*}
\]

The type rules also employ several auxiliary judgments.
\textbf{predict-routing} : \textbf{Type} × \textbf{Type} → \textbf{Type}

\[
predict-routing(\tau_o, \tau_c) = \text{strip-obs}(\tau_o) \mathbin{\tilde{n}} \tau_c
\]

\textbf{strip-obs} : \textbf{Type} → \textbf{Type}

\[
\begin{align*}
\text{strip-obs}(\text{Observe } \tau) &= \tau \\
\text{strip-obs}(\ast) &= \ast \\
\text{strip-obs}(\bigcup \tau') &= \bigcup \text{strip-obs}(\tau) \\
\text{strip-obs}(\tau) &= \bot \quad \text{otherwise}
\end{align*}
\]

\[
\tilde{n} : \textbf{Type} × \textbf{Type} → \textbf{Type}
\]

\[
\begin{align*}
\tau \tilde{n} \tau &= \tau \\
\bigcup \tau' \tilde{n} \sigma &= \bigcup \tau \tilde{n} \sigma \\
\tau \tilde{n} \bigcup \sigma' &= \bigcup \tau \tilde{n} \sigma \\
\ast \tilde{n} \tau &= \tau \\
\tau \tilde{n} \ast &= \tau \\
\text{List } \tau \tilde{n} \text{ List } \sigma &= \text{List } (\tau \tilde{n} \sigma) \\
() \tilde{n} () &= ()
\end{align*}
\]

\[
(\tau_1, \tau_2) \tilde{n} (\sigma_1, \sigma_2) = \begin{cases} 
\bot & \text{if } \tau_{11} \ll \bot \text{ or } \tau_{22} \ll \bot \\
(\tau_{11}, \sigma_{22}) & \text{if } \tau_{22} = (\sigma_{22}) \\
\text{where } \tau_{11} = \tau_1 \tilde{n} \sigma_1, \tau_{22} = (\tau_2 \tilde{n} \sigma_2)
\end{cases}
\]

\[
m(\tau_n) \tilde{n} m(\sigma_n) = \begin{cases} 
m(\sigma') & \text{if } \tau = (\sigma') \\
\bot & \text{otherwise}
\end{cases}
\]

\text{where } \tau = (\tau_n) \tilde{n} (\sigma_n)

\[
\text{Observe } \tau \tilde{n} \text{ Observe } \sigma = \begin{cases} 
\bot & \text{if } \tau' \ll \bot \\
\text{Observe } \tau' & \text{otherwise}
\end{cases}
\]

\text{where } \tau' = \tau \tilde{n} \sigma

\[
\tau \tilde{n} \sigma = \bot \quad \text{otherwise}
\]

\textbf{Figure 3.7: Key Support Metafunctions}
Definition 21. The judgment $\Gamma \vdash_p P : \tau$ checks patterns separately, which allows limiting the expressions used in patterns:

$\frac{\text{flat}(\tau)}{\Gamma \vdash_p (\$x : \tau) : (\$: \tau)}$  P-CAPTURE

$\frac{\Gamma \vdash M : \tau \quad \text{flat}(\tau)}{\Gamma \vdash_p M : \tau}$  P-EXP

$\frac{\Gamma \vdash_p P : \tau}{\Gamma \vdash_p m(P) : m(\tau)}$  P-MSG

$\frac{\Gamma \vdash_p P : \tau}{\Gamma \vdash_p \text{observe } P : \text{Observe } \tau}$  P-SUB

Definition 22. The judgment $\Gamma \vdash_{SK} \text{SK} : \tau$ checks assertion-set creation:

$\frac{}{\Gamma \vdash_{SK} \star : \star}$  SK-STAR

$\frac{\Gamma \vdash M : \tau \quad \text{flat}(\tau)}{\Gamma \vdash_{SK} M : \tau}$  SK-EXP

$\frac{\Gamma \vdash_{SK} \text{SK} : \tau}{\Gamma \vdash_{SK} (\text{SK}) : (\tau)}$  SK-PROD

$\frac{\Gamma \vdash_{SK} \text{SK} : \tau}{\Gamma \vdash_{SK} m(\text{SK}) : m(\tau)}$  SK-MSG

$\frac{\Gamma \vdash_{SK} \text{SK} : \tau}{\Gamma \vdash_{SK} \text{observe } \text{SK} : \text{Observe } \tau}$  SK-SUB

Definition 23. The judgment flat$(\tau)$ holds for types that correspond to legal assertions:

$\frac{\text{flat}(B)}{\text{flat}(\tau)}$  F-BASE

$\frac{\text{flat}(\tau)}{\text{flat}(\bigcup \tau)}$  F-UNION

$\frac{\text{flat}(\tau)}{\text{flat}(\{\tau\})}$  F-PROD

$\frac{\text{flat}(\tau)}{\text{flat}(\text{List} \tau)}$  F-LIST

$\frac{\text{flat}(\tau)}{\text{flat}(\text{Observe} \tau)}$  F-OBSERVE

$\frac{\text{flat}(\star)}{\text{flat}(\star)}$  F-STAR

Definition 24. The judgment finite$(\tau)$ describes assertion types that do not contain any uses of $\star$:
Definition 25. Rule T-PROJECT employs the \( \text{project-safe}(\tau, \sigma) \) judgment to determine if projecting a pattern of type \( \sigma \) against a set of \( \tau \)-typed assertions could yield an infinite or ill-typed result.

\[
\begin{align*}
\text{finite}(\tau) & \quad \tau \ll \sigma \quad \text{PS-CAPTURE} \\
\text{project-safe}(\tau, \xi : \sigma) & \quad \text{PS-CAPTURE} \\
\text{project-safe}(\tau, B) & \quad \text{PS-BASE} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-UNION} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-UNIONL} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-UNIONR} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-TUPLE} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-TUPLE}\star \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-MSG} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-LIST} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-SUB} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-SUB}\star \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-DISJOINT} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-DISCARD} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-UNION} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-TUPLE} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-TUPLE}\star \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-MSG} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-LIST} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-SUB} \\
\text{project-safe}(\tau, \sigma) & \quad \text{PS-SUB}\star \\
\end{align*}
\]

where

\[
\text{disjoint}(\tau, \sigma) = \tau \cap \sigma \ll: \bot
\]

Definition 26. Assertion sets \( \pi \) are given types through the judgment \( \pi \models \tau \):
\( \pi \subseteq \mathbb{Z} \quad \exists n. |\pi| = n \)
\[ \frac{\pi \vdash \text{Int}}{\pi \vdash \ast} \quad \text{M-Number} \]

\( \pi \subseteq \{ (x_i : \pi_i) \mid x_i \in \pi_i \} \quad \pi_i \vdash \tau_i \)
\[ \frac{\pi \vdash (\tau)}{\pi \vdash \text{M-Prod}} \]

\( \pi \subseteq \{ m(x_i) \mid x_i \in \pi_i \} \quad \pi_i \vdash \tau_i \)
\[ \frac{\pi \vdash m(\tau)}{\pi \vdash \text{M-Msg}} \]

\( \pi \subseteq \{ \text{observe } v \mid v \in \pi' \} \quad \pi' \vdash \tau \)
\[ \frac{\pi \vdash \text{Observe } \tau}{\pi \vdash \text{M-Observe}} \]

\( \pi \subseteq \{ \text{nil} \} \cup \{ \text{cons } x y \mid x \in \pi_1, y \in \pi_2 \} \quad \pi_1 \vdash \tau \quad \pi_2 \vdash \text{List } \tau \)
\[ \frac{\pi \vdash \text{List } \tau}{\pi \vdash \text{M-List}} \]

\( \pi = \bigcup \pi_i \quad \pi_i \vdash \tau_i \)
\[ \frac{\pi \vdash \bigcup \tau}{\pi \vdash \text{M-Union}} \]

In rule M-Union, the premise \( \pi = \bigcup \pi_i \) refers to (semantic) set union, while the conclusion \( \bigcup \tau \) uses (syntactic) type union. Furthermore, the rule does not require that the \( \pi_i \)-s are disjoint or non-empty.

**Definition 27.** The subtyping judgment \( \tau <: \sigma \) relates compatible types:
The typed language of dataspace actors satisfies a number of critical properties, most importantly, type soundness and termination.

**Theorem 28 (Soundness & Termination).** If ⊢ M : τ and error_η /∈ M, then either

1. M →^* v and ⊢ v : τ; or
2. M →^* error_{prim}.

**Interpretation** The second case of theorem 28 implies that errors may only arise due to the application of partial primitives, never through malformed communication (error_{mal}) or touching infinite sets of assertions (error_{inf}). □

**Proof of theorem 28** Starting from a well-typed term, the usual progress (lemma 29) and preservation (lemma 30) lemmas [108] show soundness.

Showing that reduction sequences terminate uses the standard “candidate” technique [45]. One salient detail of the proof is that, by the nature of assertions, the pattern variables in project are always instantiated with first-order values. □

**Lemma 29 (Progress).** If ⊢ M : τ and M ≠ error_η, then either M ∈ Val or there exists an M' such that M → M'.

3.2.4 Meta-Theorems
Proof By case analysis on the shape of M. □

Lemma 30 (Preservation). If ⊢ M : τ and M \rightarrow M', then ⊢ M' : τ.

Proof By case analysis on possible reductions M \rightarrow M'. The proof relies on several auxiliary lemmas about operations on assertion sets. In particular, lemmas 31 and 32 show that creating assertion sets never causes an error and uses of project always select finite subsets of assertions. □

Lemma 31 (Soundness of Creating Assertion Sets). If for some v and τ, ⊢ \{ v \} : AssertionSet τ, then there exists π such that make-set( v ) = π and π \models τ.

Proof By induction on the typing derivation. □

Lemma 32 (Soundness of project). If Γ ⊢ project π with v_p in M : τ, there exists an M' such that project( π, v_p, M ) = M', Γ ⊢ M' : τ.

Proof By induction on the typing derivation. □

The reduction semantics of λ_{ds} programs suffices to describe how an individual actor’s behavior function computes a response to an event. It does not explain, however, any properties of dataspace actors specified with λ_{ds}, that is, the meaning of a complete program dataspace τ_c M. Understanding how systems of actors interact requires linking with the specifics of dataspace coordination, allowing for posing and answering questions concerning actor behavior with the proper context.

3.3 COMPLETING THE MODEL

There is still one divide left to bridge: the models of dataspaces from section 2.2 and the language for programming individual actors from section 3.2 employ different notions of actor behavior functions. On one side, dataspaces appeal to abstract mathematical functions, while on the other side λ_{ds} provides concrete λ-terms, symbolic descriptions of functions given meaning through a reduction relation. This difference must be reconciled in order to combine the two semantics into a complete understanding of dataspace programs. The two boxes in figure 2.6 point out the two positions where a base language of computation interfaces with the dataspace part of the overall language.

The eval_M function for λ_{ds} (figure 3.4) provides a first step for reconciling the difference between the two parts of the model. It is a mechanism for accessing λ_{ds} terms as if they were mathematical functions. Its signature, though,

\[ eval_M : \text{Expr} \rightarrow \text{Val} + \text{error} \]

is incompatible with BehFun (figure 2.4). However, it can be used to create a behavior function. Figure 3.8 defines behavior_{ds}, which is suitable for λ_{ds} actors. The idea is to
store each actor’s private state as a pair of \( \lambda \cup ds \) values: one for the \( \lambda \)-term implementing the behavior and one for the actual private state. Invoking \( \text{eval}_M \) on a function application term comprised of the behavior function, the incoming event lowered to a \( \lambda \cup ds \) term via \([\ ]\), and the current state value yields the actor’s response to the event. Though the syntax of patches is the same in both models, the \([\ ]\) function (pronounced “lower”) is included to point out where such a translation would need to happen in an implementation. To reconcile the response, a \( \lambda \cup ds \) value, and a dataspace action, figure 3.8 also defines \([\ ]\) (pronounced “lift”), a translation that relates \( \lambda \cup ds \) values to dataspace actions. Because the syntax of events and actions in \( \lambda \cup ds \) is largely the same as that of dataspaces, the translation is straightforward. It ensures that the two sets in a patch are disjoint (n.b. figure 2.4), and performs cosmetic surgery on actor actions, inserting \( \text{behavior}_{\lambda \cup ds} \) and storing the function term in the initial state. The final metafunction from the figure, \( \text{seq} \), reconciles the \( \text{cons} \) lists computed in \( \lambda \cup ds \) with the syntactic sequences manipulated by the dataspace coordination model.
RUNNING $\lambda_{\mathcal{U}}^\mathcal{DS}$ PROGRAMS. The metafunction $\text{initialize}$ (figure 3.9) utilizes these definitions to define the meaning of a $\lambda_{\mathcal{U}}^\mathcal{DS}$ program, dataspace $\tau_c M$.

$$\begin{align*}
\text{initialize} & : \text{Init} \rightarrow \text{Config} + \text{Error} \\
\text{initialize}(\text{dataspace } \tau_c M) & = \begin{cases} 
\text{error}_{\mathcal{DS}} & \text{if } v = \text{error}_\eta \\
\text{boot}(\text{dataspace } \bar{S}) & \text{otherwise}
\end{cases} \\
& \text{where } v = \text{eval}_M(M) \\
& \quad \bar{v}_a = \text{seq}(v) \\
& \quad \bar{S} = \left\lceil \bar{v}_a \right\rceil
\end{align*}$$

Figure 3.9: $\lambda_{\mathcal{U}}^\mathcal{DS}$ Programs as Dataspaces

The function evaluates the given expression, yielding a list of actor descriptions. The seq metafunction translates the cons list to a syntactic sequence, ($\rightarrow$). Each actor value in the resulting sequence is lifted ($\left\lceil \cdot \right\rceil$) to yield an initial actor action; boot-ing the dataspace containing the actor actions yields the program’s starting configuration. At this point, reduction proceeds via $\rightarrow_{\mathcal{DS}}$.

3.3.1 Meta-Theorems

The complete operational description of $\lambda_{\mathcal{U}}^\mathcal{DS}$ dataspace systems now allows showing that the soundness theorem of section 3.2.4 generalizes to complete programs. Critically, $\lambda_{\mathcal{U}}^\mathcal{DS}$ lives up to the interface imposed on actors by the semantics of section 2.2, and dataspace routing matches the expectations encoded in the type rules of $\lambda_{\mathcal{U}}^\mathcal{DS}$ from section 3.2.3. In conjunction with a fairness property, the communication structure of a $\lambda_{\mathcal{U}}^\mathcal{DS}$ program matches static expectations.

**Theorem 33 (Fairness).** If an actor is non-quiescent, it will execute within a finite number of reductions.

As Garnock-Jones shows in his dissertation, dataspace systems are fair [40, page 59]; this permits programmers to rely on the fact that a ready actor will eventually run.

**Proof of theorem 33** The reduction semantics of dataspaces are deterministic (lemma 6) given the fixed scheduling rule. Coupled with terminating actor behaviors (lemma 39), and a round-robin scheduling policy (figure 2.6; section 2.2, definition 3), any non-quiescent actor eventually finds itself in the hole of an evaluation context. □

**Theorem 34 (System Soundness).** If $\vdash$ dataspace $\tau_c M$, then either

- $\text{initialize}(\text{dataspace } \tau_c M) = \text{error}_{\mathcal{DS}}$; or
- $\text{initialize}(\text{dataspace } \tau_c M) = \Sigma$, where
\[ \Sigma \rightarrow^*_{ds} \Sigma; \text{ or} \]
\[ \text{for all } \Sigma', \Sigma \rightarrow^*_{ds} \Sigma' \text{ implies there exits } \Sigma'' \text{ such that } \Sigma' \rightarrow_{ds} \Sigma'' \]

The soundness of \( \lambda_{ds}^{\cup} \) extends to complete dataspace programs: actors fail only due to partial primitives. That is, during communication, errors arise only from the interpretation of messages, not their shape.

**Proof of theorem 34** By inversion of \( \vdash \) dataspace \( \tau_c \) M, it must be the case that
\[ \vdash M : \text{List} (\text{Actor } \tau_c) \]

Since \( \lambda_{ds}^{\cup} \) is sound (theorem 28), either

- \( \text{eval}_M(M) = \text{error}_{prim}; \text{ or} \)
- \( \text{eval}_M(M) = v \) and \( \vdash v : \text{List} (\text{Actor } \tau_c) \).

In the first case, \( \text{initialize}(\text{dataspace } \tau_c \ M) = \text{error}_{ds} \); the program fails during startup, an acceptable outcome. Otherwise, evaluation yields a value \( v \) with a list type.

By lemma 37, \( \text{seq}(v) = \overrightarrow{v_a} \) with \( \vdash v_a : \text{Actor } \tau_c \). By inversion of the type derivation, each value in the sequence is an actor action, i.e. \( v_a = \text{actor } \tau_c, v_f, v_s \pi \).

The next step translates each \( \lambda_{ds}^{\cup} \) actor specification to a dataspace process, which according to lemma 39 is compatible with the semantics of section 2.2:

\[ \text{[actor } \tau_c, v_f, v_s \pi] = \text{actor } \text{behavior}_{\lambda_{ds}^{\cup}} (v_f, v_s) \pi \]

and boot produces an initial actor state. Finally, lemma 5 shows that such states either reduce to inertness or without end. \( \square \)

Rule T-Actor (figure 3.6) models dataspace routing by relating the assertions of interest made by an actor to the events it receives. Hence, it must be shown that it corresponds to the relationship between an individual actor’s received events and output actions across temporally distant reductions.

The relationship between events and actions relies on the fundamental theorem of dataspace event dispatch: a dataspace applies an actor’s behavior function to only those assertions in which the actor has expressed a prior interest (theorem 7, page 19). Next, dataspace routing preserves the type associated with assertion sets.

**Lemma 35** (Preservation of Types Across Dataspace Reductions). The set of assertions \( \pi \) held by a dataspace formed with communication type \( \tau_c \) has the property \( \pi \models \tau_c \) at each reduction step.

**Proof** The dataspace operations on assertion sets are set union, intersection, and subtraction, all of which preserve types. \( \square \)

The validity of T-Actor follows.
Lemma 36 (Safe Event Dispatch). If the dataspace routes a patch $\pi^+/\pi^-$ to an actor with a behavior function of type

$$ (\text{Patch } \tau_{in} \tau_{in}, \tau_{state}) \longrightarrow (\text{List } (\text{Action } \tau_{out} \tau_c), \tau_{state}) $$

then $\vdash \pi^+/\pi^- : \text{Patch } \tau_{in} \tau_{in}$. 

Proof Dataspaces compute the intersection of assertions to determine which actors to invoke on which events. The type system accounts for this intersection in rule T-ACTOR, which predicts that events delivered to the actor correspond to a type covered by the domain of the behavior function. Theorem 7 plus lemma 35 jointly verify that the semantics of dataspace routing matches this expectation. □

The function $\text{behavior} \lambda \cup ds$ is the main link connecting the two parts of the dataspace model; it relies on correctness lemmas for several helper functions.

Lemma 37 (Typed Lists Translate to Typed Sequences). If $\vdash v : \text{List } \tau$ then $\text{seq}(v) = \bar{v}_i$ and $\vdash v_i : \tau$.

Proof By routine induction. □

Lemma 38 (Typed Action Translation). If $\vdash v : \text{Action } \tau \sigma$ then there exists an $a$ such that $\lceil v \rceil = a$.

Proof Immediate from $\text{behavior} \lambda \cup ds \in \text{BehFun}$, which lemma 39 proves. □

It is now possible to show that typed actors are well-behaved, always yielding a suitable answer in response to a dispatched event.

Lemma 39 (Linking). $\text{behavior} \lambda \cup ds \in \text{BehFun}$.

Proof First, the term $M_{\text{dispatch}} = (v_f (\lceil e \rceil, v_s))$ is always well-typed. Though this property does not have a formal statement, dataspace reductions handle an actor’s private state appropriately. The behavior function is always invoked with the most recently returned, or initial, value. Therefore, the second argument of $\text{behavior} \lambda \cup ds$ is always a pair $(v_f, v_s)$. Moreover, these pairs originate from a well-typed actor term passed to $\lceil \cdot \rceil$. Also, $\text{behavior} \lambda \cup ds$ only ever updates the second element of the pair, leaving the function term $v_f$ unchanged. Thus, $v_f$ came from a well-typed actor term, and by inversion of T-ACTOR it has type

$$ (\text{Patch } \tau_{in} \tau_{in}, \tau_{state}) \longrightarrow (\text{List } (\text{Action } \tau_{out} \tau_c), \tau_{state}) $$

Since $v_s$ is either the initial state from the actor term or the second element of a pair returned by $v_f$, it must have type $\tau_{state}$. By Lemma 36, each delivered patch $\pi^+/\pi^-$, and consequently $\lceil e \rceil$, has type Patch $\tau_{in} \tau_{in}$. Therefore

$$ \vdash M_{\text{dispatch}} : (\text{List } (\text{Action } \tau_{out} \tau_c), \tau_{state}) $$

Second, by theorem 28, evaluation yields either a value of a suitable type or an error. That is, either $\text{eval}_M(M_{\text{dispatch}}) = v$ and $\vdash v : (\text{List } (\text{Action } \tau_{out} \tau_c), \tau_{state})$ or $\text{eval}_M(M_{\text{dispatch}}) = \text{error}_{\text{prim}}$. 52
• Case: \( \text{eval}_M(\text{M}_{\text{dispatch}}) = \text{error}_{\text{prim}} \). The result of \( \text{behavior}_{\lambda_{\text{ds}}} \) is then \( \text{error}_{\text{ds}} \). Since \( \text{error}_{\text{ds}} \in \text{Error} \), \( \text{behavior}_{\lambda_{\text{ds}}} \) yields a suitable answer.

• Case: \( \text{eval}_M(\text{M}_{\text{dispatch}}) = v \). By inversion, \( v = (v_{\text{list}}, v'_{\text{state}}) \) where \( \vdash v'_{\text{state}} : \tau_{\text{state}} \) and \( \vdash v_{\text{list}} : \text{List}(\text{Action} \tau_{\text{out}} \tau_{\text{c}}) \). By lemmas 37 and 38, the function produces a vector of actions: \( \text{seq}(v_{\text{list}}) \in \text{Action} \). \( \square \)

Technically, \( \text{behavior}_{\lambda_{\text{ds}}} \) is not a total function matching the signature of \( \text{BehFun} \); it is only defined on events of the expected type. Figure 2.4 requires behavior functions be total over the entire set \( \text{Event} \). However, the above properties establish that the function is only ever invoked on the portion of the space for which it is defined. Therefore, it meets the actual requirement of the dataspace semantics of always terminating with a suitable answer in response to a dispatched event.

3.4 IMPLEMENTATION

Designing a type system is an exercise in trade-offs. Every guarantee places a burden on the language implementer and a restriction on its programmers; each simplification risks losing desired precision. Implementing the proposed system as an extension to the existing Racket prototype allows exploring these choices on realistic examples within a tight design feedback loop.

In summary, the experience indicates that:

1. The type system lends itself to a straightforward implementation.

2. The restrictions are not burdensome. Typed programs may be written in much the same style as their untyped counterparts, modulo the insertion of type annotations.

3. For the programs in the design feedback loop, pattern matching on incoming assertions suffices to eliminate union types. As remarked above, occurrence typing will ease the burden even more.

DATASPACE TYPES IN ACTION. The examples and snippets from section 3.1 are written in the concrete syntax of the implementation, which extends the untyped Racket prototype. As the chat server example illustrates, the typed program in figure 3.2 is largely the same as the untyped one in figure 2.3. The major difference is that the programmer must write down the communication type, though even the design of untyped programs requires similar, informal documentation of the shape of assertions in the dataspace.

DATASPACE TYPES AS MACROS. The type checker is defined across a collection of syntax transformers that employ the types-as-macros technique [18]. In this scheme, the
expansion of a typed term produces both an elaboration—untyped syntax implementing the desired behavior—and its type. Transformers obtain the types of sub-terms through recursive traversals \[34\] and perform some analysis before finally computing their own type and elaboration. Chang’s Turnstile library manages much of the required bookkeeping, providing a notation for writing rules that resembles the standard conventions of figure 3.6.

```
(define-typed-syntax (actor τ-c beh st0 as0) ⪷
  #:fail-unless (flat-type? #τ-c)
  "Communication type must be first-order"
  [⊢ beh ⪷ beh- ⇒ (→ (Patch τ-in τ-in) τ-st
  (Instruction τ-st τ-out τ-act))]
  [⊢ st0 ⪷ st0- ⇐ τ-st]
  [⊢ as0 ⪷ as0- ⇐ (AssertionSet τ-out)]
  #:fail-unless (<: #τ-out #τ-c)
  "Actor makes assertions not allowed in this datasource"
  #:fail-unless (<: #('Actor τ-act)
  #('Actor τ-c))
  "Spawned actors not allowed in this datasource"
  #:fail-unless (<: (∩ (strip-? #τ-out) #τ-act) #τ-in)
  "Not prepared to handle all inputs"
  -----------------------------------------------
  [⊢ (untyped:actor beh- st0- as0-) ⇒ (Actor τ-c))]
```

Figure 3.10: T-Actor as a Turnstile type-and-elaboration rule

Figure 3.10 provides a representative sample, lightly edited for presentation, from the implementation. The code defines the typed \((actor \, τ-c \, beh \, st0 \, as0)\) form following the skeleton of the T-Actor rule from figure 3.6. The body (lines 2–14) is a sequence of premises, which come in two forms. The first, #:fail-unless, performs error checking. On lines 2–3, the rule checks that the given annotation sensibly describes assertions. The flat-type? procedure implements the flat judgment. If the check fails, the rule signals an error using the supplied message. The second type of premise inspects the subterms of the given syntax, utilizing Turnstile’s support of bidirectional checking \[81\] to either infer \((⇒)\) or check \((⇐)\) types. The conclusion of the rule (line 16) specifies the elaboration as the untyped actor form applied to the elaboration of the behavior, state, and assertion terms and synthesizes the type of the term. Neither the conclusion nor the premises mention the environment, \(Γ\), which is created and accessed implicitly and hygienically in this framework.

5 The Instruction type constructor is a convenience for accommodating both transition and quit actions.
A typical type system offers a number of attractive benefits: a design guide, documentation, and IDE support. Soundness adds reliability in different ways: compatibility checking of specification (type) and implementation (code), error prevention, debugging, and optimization. Thus far, no type system captured tuple-space-style communication precisely and none with soundness proofs. While the typical type system advantages accrue to this one, the error prevention and detection aspect deserves a special assessment.

In terms of error prevention, the type system provably eliminates many standard problems and three kinds of novel faults specific to the dataspace model of actor computation. (1) Types guarantee the absence of mistakes related to the shape of exchanged data, up to the expressivity of the base type system. (2) The type system tracks the structure of infinite assertion sets and statically detects when an actor selects a possibly infinite subset. Thus, the system prevents cases where data produced by a faulty component leads to the crash of a well-behaved one. (3) The presented types eliminate the possibility of actors that diverge while handling state-change notifications, an issue that thus far had to be handled outside the language.

The type system also permits the imposition of stringent constraints on specific actors. Technically speaking, actor subtyping allows a developer to place individualized constraints on the assertions of each actor in a program. By the soundness property, every assertion an actor makes falls within the range type of its behavior function. The contrapositive of this fact is useful as well: an assertion can never be made by the actor unless explicitly allowed by the range type. Using this ability, a developer can verify some aspects of well-behaved actors, such as only engaging in conversations fitting their role.

Some aspects of the dataspace communication model are beyond the expressive power of the type system. While typed actors agree on a vocabulary, \( \lambda^{\mathcal{U}}_{\mathcal{ds}} \) does not provide any further guarantee about dataspace exchanges. The type system checks only that assertions are safe to utter, but not that they contribute meaningfully to the conversation. Protocols may come with temporal constraints, requirements on the maximum (or minimum) number of related assertions that may exist, and restrictions on data beyond simple types—that an identifier is globally unique, a sequence number is greater than its predecessor, and so on. All of these properties are beyond the power of this structural type system.

Finally, this presentation has emphasized the importance of termination for actor systems. In reality, an actor that provably finishes computing after four billion years is just as problematic as one that diverges. The truly desired property for actors is responsiveness. Termination is a compromise guarantee, albeit one that is particularly useful for language models. Advances in reasoning about worst-case execution bounds [55, 68] may provide a solution to this aspect of dataspace actors.
The dataspace model of coordination and computation seeks to explore a mechanism for replicating knowledge among actors that sits between the complete isolationist approach of message sending on one end and the fully-shared memory one on the other end. In this spirit, Gelernter’s tuple spaces and the recently derived Fact Spaces have the closest resemblance to dataspaces. Hence this section compares the type systems of these latter two approaches with the one imposed on dataspace actors here. In short, integration of these models with typed programming languages has never provided a sound model of communication, as this system does.

**FACT SPACES AND AMBIENTTALK.** The Fact Spaces model [74], and especially its prototype implementation Crime, is similar to the dataspace model. Programs react to both the appearance and disappearance of facts from a shared repository. Reactions are programmed in a logic coordination language, computing new facts based on current facts, and recording (implicitly) the dependencies between facts and application actions.

The Fact Spaces model has been integrated with the AmbientTalk language [102]. AmbientTalk is a traditional Actor model [2, 52] language in the mold of E [71]. In E and AmbientTalk, objects are organized into vats; each vat runs its own event loop, dispatching incoming messages to the contained objects and running the corresponding method to completion. In addition to point-to-point messaging, AmbientTalk provides a publish/subscribe communication mechanism via topic (type) tags. Topic tags form a small nominal type system but provide no structural guarantees.

Fact Spaces and AmbientTalk are based on, and always implemented in, untyped languages. Consequently, little can be guaranteed about the behavior of programs ahead of time. Ambient contracts [95] allow, among other things, the enforcement of protocols for AmbientTalk programs. In addition, ambient contracts provide functionality related to service discovery and recovering from peer disconnections.

**TUPLE SPACES.** Linda [16, 43] introduced the tuple space model of coordination. Linda resembles a blackboard style system, where processes deposit messages in the shared space that are later read and removed by other processes. Lime [75] is an extension of the tuple space model where processes register reactions, handler functions that run on matching tuples as they appear but, notably, not as they disappear from the space.

Linda implementations with a typed base language tend to use a single Tuple type for describing items retrieved from the tuple space [62, 77, 109]. Such untyped interfaces require casting to a more specific type after read, with the potential to fail due to type mismatches. Gigaspaces [44] parameterize the type of tuple space operations such that reading a tuple does not require a cast, but do not associate a type with the entire space. Consequently, there is no assurance that the type is sensible, i.e., a tuple of that type
Blossom [47] is a tuple space implementation that fixes the type of the entire tuple space, but, being based on C++, is unsound.

**Actor and Actor-like Languages.** Point-to-point actor languages have been the subject of a number of different type systems and analyses. Though the differences in the underlying communication model makes direct comparisons uninformative, there are some similarities that may allow the results of one to carry-over to the other in a modified form.

The Conversation Calculus [106] shares notable similarities with dataspaces in its focus on multi-party interaction between anonymous components. In the conversation calculus, communication takes place within the context of a distinct, potentially nested, named conversation. A process initiates a conversation by instantiating a service name provided by another process. Much like in dataspaces, communication within a conversation is anonymous: routing finds both an attempt to send and to receive a message with the same tag within a conversation. Conversation types [12] describe the sequence of messages exchanged during each instantiation of a conversation. The type system provides flexibility in how a conversation type decomposes into multiple process types, as well as how multiple process types may merge into a conversation type, while providing the guarantee of soundness and deadlock-freedom. A significant difference that prevents their immediate application to dataspaces is the persistent nature of assertions as well as multicast-by-default communication.

The recently developed mailbox calculus and its type system [112] may also have an application to dataspaces and vice versa. Though expressive enough to describe different communication mechanisms, the mailbox calculus is tailored particularly to actor-like communication; the capability to receive from a mailbox is unique, while any number of different components may send to a mailbox. Types describe the potential contents of a mailbox with patterns, which take the form of commutative regular expressions. Like assertions, such patterns do not describe the identity of the underlying components, and grant a degree of agnosticism towards the multiplicity of a message. Checking mailbox types relies on computing the pattern describing a mailbox after a message has been received, unlike the persistent nature of assertions in a dataspace.

He et al. [50] proposed a design for Typed Akka, a more traditional message-passing actor framework for Scala. Typed Akka actors specify a particular type for the messages that they receive. While this chapter emphasizes union types for describing the assertions communicated between dataspace actors, Typed Akka seeks to work within the Scala type system. Consequently, it cannot utilize unions, even though they naturally express the underlying communication.\(^6\)

\(^6\) The next version of the language, Scala 3, is scheduled to include union types [28], opening the possibility of more expressive types for actors.
4.1 FACETS, INFORMALLY

With the facet notation, a programmer expresses an individual actor as a nest of facets. Roughly speaking, a facet groups some of the actor’s private state with the behavior related to a particular conversation. This section introduces the facet language with a running example of actors for a smart-home simulation.

```racket
(define (light-actor)
  (start-facet light-conversation
    (field [light-state : Bool ON])
    (assert (light (! light-state)))
    (on (asserted (in-room 0)) (:= light-state OFF))
    (on (retracted (in-room 0)) (:= light-state ON)))
  (spawn (light-actor)))
```

Figure 4.1 presents the code for a light-actor written in a Racket-based implementation. The actor shares its current on/off state as a light-state assertion and turns on or off based on presence sensor information in the form of in-room assertions. 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-conversation. The start-facet form imperatively extends the actor’s behavior.

A facet keeps actor-relevant state in fields. The light-conversation 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. 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 single facet of the light-actor—light-conversation—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-conversation facet, it initially makes such an assertion. Moreover, the facet keeps its assertion up to date. The light assertion in the dataspace continually reflects the current value of the actor’s light-state field. When an assertion endpoint refers to a field, the two establish a dataflow link. Thus, every time light-state changes, the actor automatically withdraws the previous light assertion from the dataspace and replaces it with an up-to-date version.

• 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 (observe (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 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, the light-state field. The dataflow mechanism automatically propagates such updates to the facet’s assertion.

The light-actor function is effectful. When called from inside an actor spawn statement (line 8), it has the effect of starting a new facet with the described behavior. In the implementation, facet operations, field creation, and endpoint creation are all free floating. Programmers may mix them 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.

Figure 4.2 visualizes the light-conversation facet of the 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.

Equipping the smart-home simulation with additional features demonstrates 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** restrict certain interactions to just the 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 4.2: Visualization of an Actor’s Facets and Facet’s Endpoints
Figure 4.3 shows how to adapt the light-actor’s implementation to this revised scenario. An unrelated difference to figure 4.1 is the use of spawn inside of the spawn-light 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 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).

```racket
(define (spawn-light [wall-switch-id : String])
  (define my-id (generate-unique-id "light"))
  (spawn
    (start-facet light-conversation
      (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 OFF)))))
```

The revised 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.\(^1\)

Here the facet operates while the assertion of \((\text{wall-switch} \ \text{wall-switch-id} \ \text{ON})\) exists in the dataspace—representing the interval of time during which the room’s switch is the on position. As the switch is flipped into the on position, the facet announces its existence and \((\text{ON})\) 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 sub-pattern $\text{room}$ binds the corresponding portion of the room-assignment assertion to the name room. Knowing which 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).

**Multi-faceted Actors.** The revised implementation of the light-actor starts several facets. Eventually it may run three concurrent facets: light-conversation and

---

\(^1\) In Racket, during is just a notational definition (macro); see section 4.1.1
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.

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-actor to configure each light. Once configured, the hub-actor both monitors and controls these lights. To accomplish these tasks, it starts a facet for each separate conversation with the various light-actors.

```scheme
(define-type-alias LightDict (Dictionary String Light))
(define (hub-actor)
  (spawn
    (start-facet hub
      (field [lights : LightDict (create-dictionary)])
      (during (light $id _)
        (on start
          (match (dict-ref (read-field lights) id #false)
            [((light room _) (control-light lights id room))
             [#false (get-room-from-user id))])))

;; auxiliary functions:
(define (get-room-from-user [id : String])
  (start-facet get-room
    (define room ... interact with user to assign a room ...)
    (stop get-room (control-light lights room))))

(define (control-light [lights : (Field LightDict)] [id : String] [room : String])
  (start-facet control
    (assert (room-assignment id room))
    (on (asserted (light id #o?))
      (set-light-state lights id room #o?))
    (on stop (set-light-state id room OFF))))

(define (set-light-state [lights : (Field LightDict)] [id : String] [room : String] [on? : Bool])
  (:= lights (dict-set (read-field lights) id (light room on?)))))
```

Figure 4.4: Implementation of the hub-actor
Figure 4.4 shows a portion of the hub-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) updates the dictionary reference for a particular light.

Using during, the actor converses with each individual light. The during statement reacts in response to a new light assertion. The “don’t care” pattern _ means the during reacts once per light rather than for each state change. 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.

To express this idea directly, the language enables reaction to facet start and stop events, respectively. That is, instead of 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 stops 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. Meanwhile, the code for cleaning up is in the control-light function (line 25). 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.

Figure 4.5 visualizes the facets of the light-actor 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.

### 4.1.1 Derived Forms

Figure 4.6 displays some derived forms. 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. The during form abstracts this pattern. Each time the asserted event fires, any pattern variables bound in pattern are instantiated to yield a concrete assertion, pattern’. The (on (retracted pattern’) ...) endpoint monitors this assertion in the new facet.

Figure 4.6 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 (exceptional) failure. While an exception in during tears down the current actor, a failure in during/spawn terminates the separate actor but leaves the spawning one alone.

\[(\text{during pattern during-body ...}) \equiv (\text{on (asserted pattern)} \begin{array}{l}
\text{start-facet theBody} \\
\text{on (retracted pattern')} \\
\text{stop theBody}) \\
\text{during-body ...})\]  

\[(\text{define/query-value id expr \_0 pattern expr}) \equiv (\text{field \_id [id expr]}\_0) \begin{array}{l}
\text{on (asserted pattern)} (\_id := \_id expr) \\
\text{on (retracted pattern)} (\_id := \_id expr)\_0)\]  

\[(\text{define/query-set id pattern expr}) \equiv (\text{field \_id : (Set \_T) empty-set}) \begin{array}{l}
\text{on (asserted pattern)} \\
\text{on (retracted pattern)}
\end{array} (\text{set-add (\_id expr)}) (\text{set-remove (\_id expr)})\]
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 4.6 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 a set of 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 that may or may not have an instance in the dataspace. For example, the smart home program can be extended 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

\[
\text{(define/query-value config DC (configuration $lights ...))}
\]

\[
\text{(configuration lights ...))}
\]

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. A user-interface component for the smart home may use it to show the user all of the lights currently installed in the system:

\[
\text{(define/query-set lights (light $id _) id)}
\]

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

4.1.2 The Synergy of Facets

The smart-home example illustrates several strengths of the facet notation:

- Events are automatically demultiplexed. In the excerpts from the example, no lines of code are dedicated to passing events from one place to another. Instead,
event-handling code is described in tandem with relevant events, and the language implementation takes care of the rest.

- Dataflow keeps public and private state automatically in sync (figure 4.2).
- Control is composable in both breadth and depth (figure 4.5).
- Facets, fields, and event handlers compose naturally with the rest of the language. A programmer may abstract over expressions of these features as needed, define those in one module, export them to another, and so on (figure 4.4).
- The core facet language is powerful enough to build useful abstractions (figure 4.6).
- State is highly localized. Fields can be created in any facet, even deeply nested ones. Multiple instantiations of the same facet description lead to multiple instantiations of the same field description.

Generally put, the facet notation combines 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 [9] and CRIME [73]. Dataflow exists in many forms [27]. Updates to behavior have been around as long as the Actor model [53]. 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.

4.2 FACETS, FORMALLY

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 latter deserves a formal semantics; section 4.2.5 provides the appropriate definitions in the style of section 3.3 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 4.2.1 and 4.2.2). 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.

4.2.1 Formal Syntax

Figure 4.7 defines the abstract syntax of facet-oriented actors. Statements Pr imperatively update the actor’s state in one of several ways:
• **start fn A \(\rightarrow D Pr\)**—starting a facet with name fn, assertions A, and event handlers \(\rightarrow D Pr\);

• **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, plus propagation; and

• **spawn Pr**—spawning an additional actor.

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 (\(\emptyset\)).

---

**Figure 4.7: Program Syntax**
The set of expressions comprises

- $b$—basic values;
- $p(\vec{e})$—invocations of potentially partial primitive operations;
- $x$—references to variable names;
- $!x$—field access; and
- $\text{observe } e, m(\vec{e})$—assertion constructors.

A subset of expressions known to be total forms the category $e_v$.\(^3\)

A union ($A$) of assertion templates ($k$) defines the assertions made by a facet. Templates describe families of assertions built with labels ($m$), interests ($\text{observe}$), wildcards ($\star$), and expressions ($e_v$).

Event handlers consist of an event description ($D$) and a body ($Pr$). Events describe occurrences that are either internal to the actor ($\text{start}$, $\text{stop}$) or particular assertions in the dataspace ($\text{asserted P}$, $\text{retracted P}$). In the latter case, the description includes a pattern $P$ defining the relevant assertions. Patterns include match-anything wildcards ($\star$), binding variables ($x : \tau$), and equality to a run-time value ($= e_v$), as well as assertion constructors ($m$, $\text{observe}$).

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 in one place.
- 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 $\text{struct}$ names.
- Expressions to be matched are noted as $= e_v$.
- Binding variables in patterns require a type annotation.

Here is the basic light-actor from figure 4.1 in the model’s syntax:

```plaintext
field state = ON in
    start light-conversation
    light(!state) \cup \emptyset
    (asserted in-room(count: Int)
    state := if(equal?(count, 0), OFF, ON))
```

\(^3\) To keep the meta-theory simple, $e_v$ does not include any applications of primitive operations. Extending this category with total operations, such as addition and logical negation, would be straightforward. Extending it with partial operations would severely complicate the semantics without any gains in insight.
4.2.2 Semantics

Since individual actors consume events and produce assertions, the semantics of facet-oriented actors is specified 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 (\( \Delta \)) or outputs in the form of spawned actors (\( \text{Pr} \)). An unlabeled transition relation (\( \cdot \)), represents actor-internal work. Patches \( \Delta \) comprise of two disjoint assertions sets \( \pi \), representing added and removed assertions. Assertions \( c \) range over basic values, labeled tuples, and assertions of interest.

The formulation of machines state demands additional syntax. Figure 4.8 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, \( \langle \text{FT}; \quad I; \quad \text{PS}; \quad \pi; \quad \sigma \rangle \), consists of five components:

- **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: \( \text{FT} \). A node \( \text{fn} \mid \langle A(\Delta\text{Pr}) \rangle \text{FT} \) 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 \( \epsilon \).

- **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 \( \text{PS} \). A pending script \( (\text{fid}, \text{Pr}) \) is a sequence of statements to be performed and pertinent context information. The context is a facet ID \( \langle \text{fn} \rangle \) 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 designated location in the tree; stop a running one; and spawn an actor.

- **the information known to the actor**
  Specifically, the store \( \sigma \) holds the contents of the fields, while the set of assertions \( \pi \) keeps track of the actor’s currently known assertions. The assertions are needed to boot new facets, because an event handler may describe an assertion the actor has already seen.\(^4\)

Finally, there are some miscellaneous pieces of syntax. An Evt is either a patch or facet start/stop. Evaluation reduces expressions to values \( v \) and patterns to pattern values \( P_v \), which can be matched against incoming events. An interpretation function \( \delta \) gives semantics to primitive operations; substitutions \( \gamma \) are represented by finite maps.

---

\(^4\) Dataspaces track the set of assertions known to an actor rather than each facet.
Figure 4.8: Evaluation Syntax

from variables to values; and matching a pattern against a set of assertions yields a set of substitutions $S$.

Figure 4.9—above the horizontal line—shows the initial machine state $M$ for the light-conversation facet from above. The rest of the figure displays the first two transitions of the machine.

**Transition Relation.** Figure 4.10 defines the transition relation on machine states. It describes the behavior of a single facet-based actor in response to incoming events. The semantics ignores how such events arrive; section 2.2 covers that element. The notation $M \rightarrow M'$ is a shorthand for $M \xrightarrow{\text{trans}} M'$. Section 4.2.3 provides the full definition of the meta-functions mentioned by figure 4.10.

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
\[
(l-c[EH]; ; ; \emptyset; \{ \text{state} \rightarrow \text{ON} \})
\]
where \( f_{l-c} = \text{light-conversation} \)

\[
l-c[X] = f_{l-c}[A_{l-c} X] \epsilon
\]

\[
A_{l-c} = \text{light}(!\text{state}) \cup \emptyset
\]

\[
EH = (\text{asserted in-room(count : Int)}\)
\]

\[
\begin{align*}
\text{state} & : = \text{if(equal?(count, 0), OFF, ON)} \\
\end{align*}
\]

\[
\langle l-c[EH]; ; ; ; \{ \text{in-room(0)} \}; \{ \text{state} \rightarrow \text{ON} \} \rangle
\]

(by inject)

\[
\langle l-c[EH]; ; \text{PS}; \{ \text{in-room(0)} \}; \{ \text{state} \rightarrow \text{ON} \} \rangle
\]

where \( \text{PS} = (f_{l-c}, \text{state} : = \text{if(equal?(0, 0), OFF, ON)} \)

(by transfer)

\[
\langle l-c[EH]; ; ; ; \{ \text{in-room(0)} \}; \{ \text{state} \rightarrow \text{OFF} \} \rangle
\]

Figure 4.9: The light-conversation facet as an initial machine state and its transitions

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. The dispatch metafunction (definition 41) 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; definition 44) 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. The actor does not run any stop event
\[(FT; \; \cdot; \; \pi; \; \sigma) \xrightarrow{\text{ExtEvt}} (FT; \; \cdot; \; P_S; \; \pi'; \; \sigma') \quad \text{(inject)}\]

where
\[
\pi' = \pi \oplus \text{ExtEvt} \quad P_S = \text{dispatch}(FT, \text{ExtEvt, } \pi, \pi', \sigma, \langle \cdot \rangle) \]

\[\langle FT; \; \cdot; \; (\text{fid, Pr}) \cdot P_S; \; \pi'; \; \sigma' \rangle \xrightarrow{\text{(transfer)}} M' \]

where
\[
M' = \begin{cases} (FT; \; T; \; P_S; \; \pi; \; \sigma') & \text{if } T, \sigma' = \text{p-e(Pr, } \sigma, \text{ fid)} \\
\text{error} & \text{otherwise} \end{cases} \]

\[\langle FT; \text{start fn } A (D \text{ Pr}) \odot \text{fid} \cdot T; \; P_S; \; \pi; \; \sigma \rangle \xrightarrow{\text{(start)}} \langle FT'; \; T; \; P_{\text{new}}; \; \pi; \; \sigma \rangle \quad \text{(start)}\]

where
\[
\begin{align*}
\text{fn}_{\text{new}} & \text{ fresh in FT} \\
\text{FT}_{\text{new}} & = \text{fn}_{\text{new}}[A \langle D \text{ Pr} \odot \text{fn} \mapsto \text{fn}_{\text{new}} \rangle], \epsilon \\
\text{PS}_{\text{start}} & = \text{dispatch}(\text{FT}_{\text{new}}, \text{start, } \emptyset, \emptyset, \sigma, \text{ fid}) \\
\text{PS}_{\text{boot}} & = \text{dispatch}(\text{FT}_{\text{new}}, \pi / \emptyset, \emptyset, \pi, \sigma, \text{ fid}) \\
\text{FT}' & = \begin{cases} C[\text{FT}_{\text{new}}] & \text{if locate(FT, fid) = C} \\
\text{FT} & \text{otherwise} \end{cases} \\
\text{PS}_{\text{stop}} & = \begin{cases} \ldots & \text{if locate(FT, fid) defined} \\
\text{dispatch}(\text{FT}_{\text{new}}, \text{stop, } \pi, \pi, \sigma, \text{ fid}) & \text{otherwise} \end{cases} \]
\]

\[\text{begin} \quad \text{PS}_{\text{new}} \xrightarrow{\text{}} \text{PS}_{\text{start}} \xrightarrow{\text{}} \text{PS}_{\text{boot}} \xrightarrow{\text{}} \text{PS}_{\text{stop}} \]

\[\langle FT; \text{stop fn } \cdot \; I; \; P_S; \; \pi; \; \sigma \rangle \xrightarrow{\text{(stop)}} \langle FT'; \; I; \; P_S \cdot \text{PS}_{\text{stop}}; \; \pi; \; \sigma \rangle \quad \text{(stop)}\]

where
\[
\begin{align*}
\text{FT}' & = \begin{cases} C[\epsilon] & \text{if FT = C[\text{fn} \ldots, FT]} \\
\text{FT} & \text{otherwise} \end{cases} \\
\text{fid} & = \text{facet-context}(C) \\
\text{PS}_{\text{stop}} & = \begin{cases} \text{dispatch}(\text{fn} \ldots, FT, \text{stop, } \pi, \pi, \sigma, \text{ fid}) & \text{if FT = C[\text{fn} \ldots, FT]} \\
\ldots & \text{otherwise} \end{cases} \]
\]

\[\langle FT; \text{spawn Pr } \cdot \cdot \; I; \; P_S; \; \pi; \; \sigma \rangle \xrightarrow{\text{Pr}} \langle FT; \; I; \; P_S; \; \pi; \; \sigma \rangle \quad \text{(spawn)}\]

Figure 4.10: Transition
handlers in the case of a crash. Since the state of the actor could be inconsistent, with fields partially updated, orderly shutdown may not be possible.

**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 (definition 41) 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 (definition 45) constructs the appropriate context. It is possible that no such context exists. This circumstance occurs when the desired parent, or one of its ancestors, 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 further work to do.

**spawn** —The facet wishes to create an actor, which the machine treats as an output instruction to the surrounding dataspace. Formally, the transition comes with a \( PT \) label.

The top-most machine state in figure 4.9 transitions in two steps to an inert state. The first one injects the script, consuming the patch \( \{\text{in-room}(0)\} / \emptyset \) as the label indicates. The second enqueues the script’s instruction and performs it.

### 4.2.3 Metafunctions

The semantics of section 4.2.2 refers to several metafunctions. This section provides their formal definitions.

**Definition 40** (\( boot_{PT} \)). The \( boot_{PT} \) 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.

\[
\text{boot}_{\Pr} : \Pr \times \sigma \xrightarrow{\text{partial}} M
\]

\[
\text{boot}_{\Pr}(\text{field } x = e \ \text{in } \Pr, \sigma) = \text{boot}_{\Pr}(\Pr[x \mapsto x'], \sigma[x' \mapsto v])
\]

if \( v = \text{eval}(e, \sigma) \)

where

\( x' \) fresh in \( \sigma, \Pr \)

\[
\text{boot}_{\Pr}(\text{let } x = e \ \text{in } \Pr, \sigma) = \text{boot}_{\Pr}(\Pr[x \mapsto v], \sigma)
\]

if \( v = \text{eval}(e, \sigma) \)

\[
\text{start fn } A (\Pr, \sigma) = \langle \text{fn root }[\emptyset]. FT; \cdot \Pr; \emptyset; \sigma \rangle
\]

where

\( FT = \text{fn } A (\Pr)[.e \Pr] \)

\( \Pr = \text{dispatch}(FT, \text{start}, \emptyset, \emptyset, \sigma, \langle \text{fn root} \rangle) \)

\( \text{fn root} \) free in \( FT \)

**Definition 41 (dispatch).** The dispatch function matches an event against each handler in a tree of facets, accumulating the successes as a list of pending scripts.

\[
\text{dispatch} : FT \times \text{Evt} \times \pi \times \pi' \times \sigma \times \text{fid} \xrightarrow{\text{partial}} PS
\]

\[
\text{dispatch}(e, \text{Evt}, \pi, \pi', \sigma, \text{fid}) = \cdot
\]

\[
\text{dispatch}(FT, \text{Evt}, \pi, \pi', \sigma, \text{fid}) = \text{dispatch}_1(\langle \text{fn ctx } \cdot \text{fn} \rangle, D, \Pr, \text{Evt}, \pi, \pi', \sigma)
\]

\[
\cdot \text{dispatch}(FT', \text{Evt}, \pi, \pi', \sigma, \langle \text{fn ctx } \cdot \text{fn} \rangle)
\]

where

\( FT = \text{fn } A (\Pr)[.e \Pr] \)

\( \Pr = \text{dispatch}(FT, \text{start}, \emptyset, \emptyset, \sigma, \langle \text{fn root} \rangle) \)

\( \text{fn root} \) free in \( FT \)

It utilizes a helper function, \( \text{dispatch}_1 \), to instantiate the body of a single event handler with the substitution(s) arising from matching against a single event.

\[
\text{dispatch}_1 : \text{fid} \times D \times \Pr \times \text{Evt} \times \pi \times \pi' \times \sigma \xrightarrow{\text{partial}} PS
\]

\[
\text{dispatch}_1(\text{fid}, D, \Pr, \text{Evt}, \pi, \pi', \sigma) = \begin{cases} 
(\text{fid}, \text{unroll}(m)) & \text{if } \exists n. n > 0, |S| = n \\
\cdot & \text{if } |S| = 0
\end{cases}
\]

where

\( S = \text{match}_D(D, \text{Evt}, \pi, \pi', \sigma) \)

\( m = \{ \gamma(\Pr) \mid \gamma \in S \} \)

The partiality of \( \text{dispatch} \) and \( \text{dispatch}_1 \) arises from the possibility that project (definition 42) yields an infinite set of matches.
Finally, \textit{unroll} sequentializes a set of statements using an arbitrary yet fixed ordering.

\[
\text{unroll} : \mathcal{P}(\mathcal{P}r) \rightarrow \mathcal{P}r
\]

\[
\text{unroll}(\emptyset) = \emptyset
\]

\[
\text{unroll}(\{\mathcal{P}r\} \uplus m) = \mathcal{P}r; \text{unroll}(m)
\]

\textbf{Definition 42 (Matching Events).} The \textit{matchD} 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.

\[
\text{matchD} : \mathcal{D} \times \mathcal{E} \times \pi \times \pi' \times \sigma \xrightarrow{\text{partial}} \mathcal{S}
\]

\[
\text{matchD}(\text{start}, \text{start}, \pi, \pi', \sigma) = \{ \emptyset \}
\]

\[
\text{matchD}(\text{stop}, \text{stop}, \pi, \pi', \sigma) = \{ \emptyset \}
\]

\[
\text{matchD}(\text{asserted } \mathcal{P}, \pi^+ / \pi^-, \pi, \pi', \sigma) = \text{project}(P_v, \pi, \pi', \pi^+)
\]
\[
\text{if } P_v = \text{eval}(P, \sigma)
\]

\[
\text{matchD}(\text{retracted } \mathcal{P}, \pi^+ / \pi^-, \pi, \pi', \sigma) = \text{project}(P_v, \pi, \pi', \pi^-)
\]
\[
\text{if } P_v = \text{eval}(P, \sigma)
\]

The \textit{project} function handles matching a pattern against a set of assertions, yielding a set of substitutions. The set is empty when there are no matching assertions in the set.

\[
\text{project} : \mathcal{P}_v \times \pi \times \pi \times \pi \rightarrow \mathcal{S}
\]

\[
\text{project}(P_v, \pi, \pi', \pi) = \{ \text{match}(P_v, I) \mid I \in \{ \text{inst}(P_v, c) \mid c \in \pi \},
\]

\[
\text{known}(I, \pi_i) \neq \text{known}(I, \pi'_i) \}
\]

where

\[
\text{known}(I, \pi) = 1 \text{ if } \exists c \in \pi \text{.match}(I, c) \text{ defined; else, } 0
\]

The helper \textit{inst} partially matches a pattern against a value, leaving wildcards and binders in place.

\[
\text{inst} : \mathcal{P}_v \times c \xrightarrow{\text{partial}} \mathcal{P}_v
\]

\[
\text{inst}(*, c) = *
\]

\[
\text{inst}(x : \tau, c) = x
\]

\[
\text{inst}(= \nu, \nu) = \nu
\]

\[
\text{inst}(m(\overrightarrow{P}_v), m(\overrightarrow{c})) = m(\overrightarrow{P}_v)
\]
\[
\text{if } |\overrightarrow{P}_v| = |\overrightarrow{c}|
\]

\[
\overrightarrow{P}'_v = \text{inst}(P_v, c)
\]

\[
\text{inst}((\text{observe } P_v, \text{observe } c) = \text{observe } P'_v \text{ if } P'_v = \text{inst}(P_v, c)
\]
The `matchP` function implements basic value-against-pattern matching. The definition assumes without loss of generality that all binders in a given pattern are unique.

\[
\begin{align*}
\text{matchP} & : P \times c \xrightarrow{\text{partial}} \gamma \\
\text{matchP}(\ast, c) &= \emptyset \\
\text{matchP}(x : \tau, c) &= \{x \mapsto c\} \\
\text{matchP}(= v, v) &= \emptyset \\
\text{matchP}(m(P_v), m(c)) &= \bigcup \gamma \\
& \quad \text{if } |P_v| = |c| \\
& \quad \gamma = \text{matchP}(P_v, c)
\end{align*}
\]

**Definition 43 (Patch Incorporation).** The \(\oplus\) operation updates an actor’s knowledge of assertions based on a patch.

\[
\oplus : \pi \times \Delta \longrightarrow \pi \\
\pi \oplus \pi^+ / \pi^- = (\pi \cup \pi^+) - \pi^-
\]

**Definition 44 ((Partial) Evaluation).** The `p-e` function partially-evaluates a script, yielding a sequence of instructions and an updated store.

\[
\begin{align*}
p-e & : Pr \times \sigma \times \text{fid} \xrightarrow{\text{partial}} I \times \sigma \\
p-e(\emptyset, \sigma, \text{fid}) &= \cdot \\
p-e(\text{Pr}_1; \text{Pr}_2, \sigma, \text{fid}) &= \overrightarrow{I_1} \cdot \overrightarrow{I_2} \cdot \sigma'' \\
& \quad \text{if } \overrightarrow{I_1} \cdot \sigma' = p-e(\text{Pr}_1, \sigma, \text{fid}) \\
& \quad \overrightarrow{I_2} \cdot \sigma'' = p-e(\text{Pr}_2, \sigma', \text{fid}) \\
p-e(\text{start fn } A [\overrightarrow{\text{D Pr}}], \sigma, \text{fid}) &= \text{start fn } A [\overrightarrow{\text{D Pr}}] @ \text{fid}, \sigma \\
p-e(\text{stop fn } \text{Pr}, \sigma, (\overrightarrow{\text{fn fn}})) &= \text{stop fn } \overrightarrow{I}, \sigma' \\
& \quad \text{if } p-e(\text{Pr}, \sigma, (\overrightarrow{\text{fn}})) = \overrightarrow{I}, \sigma' \\
p-e(\text{spawn } \text{Pr}, \sigma, \text{fid}) &= \text{spawn } \text{Pr}, \sigma
\end{align*}
\]

\[
\begin{align*}
p-e(\text{let } x = e \text{ in } \text{Pr}, \sigma, \text{fid}) &= p-e(\text{Pr}[x \mapsto v], \sigma, \text{fid}) \text{ if } v = \text{eval}(e, \sigma) \\
p-e(\text{field } x = e \text{ in } \text{Pr}, \sigma, \text{fid}) &= p-e(\text{Pr}', \sigma', \text{fid}) \\
& \quad \text{if } v = \text{eval}(e, \sigma) \\
& \quad \text{where } \\
& \quad x' \text{ fresh in } \sigma, \text{Pr} \\
& \quad \sigma' = \sigma[x' \mapsto v] \\
& \quad \text{Pr}' = \text{Pr}[x \mapsto x'] \\
p-e(x := e, \sigma \cup \{x \mapsto v\}, \text{fid}) &= \sigma[x' \mapsto v'] \\
& \quad \text{if } v' = \text{eval}(e, \sigma)
\end{align*}
\]
The evaluator for expressions is \( \text{eval} \).

\[
\text{eval} : \ e \times \sigma \xrightarrow{\text{partial}} v \\
\text{eval}(b, \sigma) = b \\
\text{eval}(! x, \sigma) = \sigma(x) \\
\text{eval}(p(e), \sigma) = \delta(p, \overrightarrow{v}) \text{ if } \overrightarrow{v} = \text{eval}(e, \sigma) \\
\text{eval}(m(e), \sigma) = m(\overrightarrow{v}) \text{ if } \overrightarrow{v} = \text{eval}(e, \sigma) \\
\text{eval}(\text{observe } e, \sigma) = \text{observe } v \text{ if } v = \text{eval}(e, \sigma)
\]

The function \( \text{eval}_P \) evaluates a pattern to a pattern value. Its partiality is due to the possibility of unbound field names. For well-typed programs, \( \text{eval}_P \) is total.

\[
\text{eval}_P : P \times \sigma \xrightarrow{\text{partial}} P_v \\
\text{eval}_P(*) = * \\
\text{eval}_P(x : \tau, \sigma) = x : \tau \\
\text{eval}_P(\sigma_1 = \sigma_2, \sigma) = \text{eval}(\sigma_1, \sigma_2) \\
\text{eval}_P(m(\overrightarrow{v}), \sigma) = m(\overrightarrow{v}) \text{ if } \overrightarrow{v} = \text{eval}_P(P, \sigma) \\
\text{eval}_P(\text{observe } P, \sigma) = \text{observe } v \text{ if } v = \text{eval}_P(P, \sigma)
\]

**Definition 45 (Facet Contexts).** The \( \text{facet-context} \) function produces the path of facet names (a fid) leading to the hole in a context.

\[
\text{facet-context} : C \rightarrow \text{fid} \\
\text{facet-context}(\Box) = \langle \cdot \rangle \\
\text{facet-context}(\text{fn}[A(\overrightarrow{D}d)], F^1 \cdot C \cdot F^2) = \langle \text{fn} \cdot \text{fn} \rangle \\
\text{where} \langle \text{fn} \rangle = \text{facet-context}(C)
\]

Meanwhile, \( \text{locate} \) operates in the reverse direction, producing the context for a particular fid in a tree of facets.

\[
\text{locate} : \ FT \times \text{fid} \xrightarrow{\text{partial}} C \\
\text{locate}(FT, \langle \cdot \rangle) = \Box \\
\text{locate}(\text{fn}[A(\overrightarrow{D}d)], F^1, \langle \text{fn} \cdot \text{fn} \rangle) = \text{fn}[A(\overrightarrow{D}d)], F^1 \cdot C \cdot F^2 \text{ if } \text{locate}(F^1, \langle \text{fn} \rangle) = C \\
F^1 = F^1_1 \cdot F^1_2 \cdot F^2_2
\]

**Definition 46 (Actor Assertions).** The \( \text{assertions-of} \) function produces the current assertions made by an actor given its active facet tree and field store.

\[
\text{assertions-of} : \ FT \times \sigma \rightarrow \pi \\
\text{assertions-of}(e, \sigma) = \varnothing \\
\text{assertions-of}(\text{fn}[A(\overrightarrow{D}d)], F^1, \sigma) = \text{assertions-of}_{A}(A, \sigma) \cup \\
\cup \text{assertions-of}_{D}(D, \sigma) \cup \\
\cup \text{assertions-of}_{\text{ft}}(F, \sigma)
\]
It utilizes a family of helper functions for determining the assertions associated with endpoints. The $\text{assertions-of}_A$ and $\text{assertions-of}_k$ functions do the work for assertion endpoints.

\[
\begin{align*}
\text{assertions-of}_A & : A \times \sigma \rightarrow \pi \\
\text{assertions-of}_A(\emptyset, \sigma) & = \emptyset \\
\text{assertions-of}_A(k \cup A, \sigma) & = \text{assertions-of}_k(k, \sigma) \cup \text{assertions-of}_A(A, \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{assertions-of}_k & : k \times \sigma \rightarrow \pi \\
\text{assertions-of}_k(\star, \sigma) & = \text{Assertion} \\
\text{assertions-of}_k(m(k_i), \sigma) & = \{ m(c_i) \mid c_i \in \text{assertions-of}_k(k_i, \sigma) \} \\
\text{assertions-of}_k(\text{observe } k, \sigma) & = \{ \text{observe } c \mid c \in \text{assertions-of}_k(k, \sigma) \} \\
\text{assertions-of}_k(\text{ev}_c, \sigma) & = \{ \text{eval}(c, \sigma) \}
\end{align*}
\]

While the $\text{assertions-of}_D$ and $\text{assertions-of}_P$ produce the assertion of interest when needed by an event-handler endpoint:

\[
\begin{align*}
\text{assertions-of}_D & : D \times \sigma \rightarrow \pi \\
\text{assertions-of}_D(\text{start}, \sigma) & = \emptyset \\
\text{assertions-of}_D(\text{stop}, \sigma) & = \emptyset \\
\text{assertions-of}_D(\text{asserted } P, \sigma) & = \text{assertions-of}_P(P, \sigma) \\
\text{assertions-of}_D(\text{retracted } P, \sigma) & = \text{assertions-of}_P(P, \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{assertions-of}_P & : P \times \sigma \rightarrow \pi \\
\text{assertions-of}_P(\star, \sigma) & = \text{Assertion} \\
\text{assertions-of}_P(x : \tau, \sigma) & = \text{Assertion} \\
\text{assertions-of}_P(m(F_i), \sigma) & = \{ m(c_i) \mid c_i \in \text{assertions-of}_P(F_i, \sigma) \} \\
\text{assertions-of}_P(\text{observe } P, \sigma) & = \{ \text{observe } c \mid c \in \text{assertions-of}_P(P, \sigma) \} \\
\text{assertions-of}_P(= e_v, \sigma) & = \{ \text{eval}(e_v, \sigma) \}
\end{align*}
\]

### 4.2.4 Machine Transition Example

The full implementation of the `light-actor` in figure 4.3 is a good example for illustrating how the facet machine deals with loading and manipulating states. Figure 4.11 shows the full version of the actor description, assuming suitable bindings for `wall-switch-id` and `my-id`.

Figure 4.12 shows the complete machine transition sequence, starting from the initial state.

- From there, the machine executes a labeled inject transition receiving a `wall-switch` assertion. The transition enqueues a script with the body of the event handler. Next, a transfer evaluates the script, yielding a `start` instruction. The next transition performs the instruction, inserting the new facet in the tree. At this point, the actor is inert again and ready for another event.
\[ LC = \]
start light-conversation
\[ \emptyset \]
\[ EH_{WS} \]

\[ EH_{WS} = \]
(asserted wall-switch(wall-switch-id, ON)
\[ Pr_{WS} \]

\[ Pr_{WS} = \]
field state = ON in
start during<wall-switch>
light(my-id, !state) \cup \emptyset
\[ EH_{dws} \]

\[ EH_{dws} = \]
(retracted wall-switch(wall-switch-id, ON)
stop during<wall-switch>)
(asserted room-assignment(my-id, room: String)
\[ Pr_{RA} \]

\[ Pr_{RA} = \]
start during<room-assignment>
\[ \emptyset \]
\[ EH_{drs} \]

\[ EH_{drs} = \]
(retracted room-assignment(my-id, room)
stop during<room-assignment>)
(asserted in-room(0)
state := OFF)
(retracted in-room(0)
state := ON)

Figure 4.11: The full light-actor

- There are two enabled events, the retraction of the wall-switch assertion or the appearance of a room-assignment. This example shows the occurrence of the latter via another inject transition. In much the same manner as before, the event yields a sequence of transfer and start transitions that update the facet tree. Since the room-assignment event handler binds the name of the room, all references to the name room are substituted for the concrete value in the assertion—here, "kitchen”.

- After reaching inertness again, the actor has numerous enabled events. The example shows an event representing the retraction of the wall-switch assertion.
Figure 4.12: The light-conversation facet as an initial machine state and its transitions.
The resulting script and instruction stop the facet implementing the \texttt{ON} behavior, returning the facet tree to the initial state.

### 4.2.5 Dataspace Programs

The following definitions link the semantics of faceted actors to that of dataspace programs from section 2.3. The structure is much the same as that of section 3.3.

**Definition 47 (\texttt{boot}_{DS}).** The \texttt{boot}_{DS} metafunction initializes a dataspace program based on a collection of initial facet-based actor descriptions.

\[
\text{\texttt{boot}}_{\text{DS}} : \text{\texttt{Pr}} \rightarrow \text{DS} \\
\text{\texttt{boot}}_{\text{DS}}(\text{\texttt{spawn P}}) = [(\ell, \pi/\emptyset); \emptyset; \vec{X}]
\]

where

\[
\vec{P} = \text{\texttt{boot}}_{\text{P}}(\text{\texttt{Pr}}) \\
\vec{\Sigma}, \vec{\pi} = \text{\texttt{boot}}_{\Sigma}(\vec{P}) \\
\ell = 0 \cdot 1 \ldots |\text{\texttt{Pr}}| - 1 \\
\vec{X} = \ell \mapsto \Sigma
\]

The \texttt{boot}_{P} metafunction translates a facet-level \texttt{spawn} to a dataspace process description, \texttt{P}:

\[
\text{\texttt{boot}}_{\text{P}} : \text{Pr} \rightarrow \text{P} \\
\text{\texttt{boot}}_{\text{P}}(\text{Pr}) = \text{actor interp}_{\text{M}} M_{\text{boot}} \text{ assertions-of}_{\text{M}}(M_{\text{boot}})
\]

where \(M_{\text{boot}} = \begin{cases} 
M & \text{if } \text{\texttt{boot}}_{\text{P}}(\text{Pr}, \emptyset) = M \\
\text{error} & \text{otherwise}
\end{cases} \)

**Definition 48 (\texttt{interp}_{\text{M}}).** The \texttt{interp}_{\text{M}} function implements behavior for facet-based actors compatible with the dataspace interface.

\[
\text{\texttt{interp}}_{\text{M}} : \text{pe} \times M \rightarrow M \times \text{act} \\
\text{\texttt{interp}}_{\text{M}}(\text{pe}, \text{error}) = (\text{error}, \cdot) \\
\text{\texttt{interp}}_{\text{M}}(\text{pe}, M) = \begin{cases} 
(\text{error}, \text{patch}(M, \text{error})) & \text{if } M \xrightarrow{\text{pe}} M' \xrightarrow{\ell, \cdot} \text{error} \\
(M'', \text{patch}(M, M'') \cdot \text{label-action}(\ell)) & \text{if } M \xrightarrow{\text{pe}} M' \xrightarrow{\ell, \cdot} M'', \text{inert}(M'')
\end{cases}
\]

The helper function \texttt{patch} calculates the difference in assertions between two machine states:

\[
\text{\texttt{patch}} : M \times M \rightarrow \Delta \\
\text{\texttt{patch}}(M, M') = \frac{\text{assertions-of}_{\text{M}}(M') - \text{assertions-of}_{\text{M}}(M)}{\text{assertions-of}_{\text{M}}(M) - \text{assertions-of}_{\text{M}}(M')}
\]
The label-action function translates a facet machine’s internal transition label to a sequence of (zero or one) actions.

\[
\text{label-action} : I \to act
\]

\[
\text{label-action}(\cdot) = \cdot
\]

\[
\text{label-action}(\Pr) = \text{boot}_P(Pr)
\]

**Definition 49** (assertions-of\(_M\)). The assertions-of\(_M\) function extends the family of functions from definition 46, determining the active assertions of a facet machine.

\[
\text{assertions-of}_M : M \to \pi
\]

\[
\text{assertions-of}_M(\text{error}) = \emptyset
\]

\[
\text{assertions-of}_M(\langle FT; \xrightarrow{-} I; \xrightarrow{-} PS; \pi; \sigma \rangle) = \text{assertions-of}(FT, \sigma)
\]

### 4.3 Type Checking Facets

Figure 4.13 defines the syntax of types. Types \(\tau\) summarize the result of expressions, with base types \(B\) describing primitive values. Template types \(\tau_k\) correspond to assertion templates \(k\), roughly the sorts of things that may be found in the dataspace, including wildcards \(*\). Effect types \(\text{Eff}\) correspond to communication actions: the potential to output some type of assertion, \(\text{output} \ \tau_k\); constraining the types of input events, \(\text{Input} \ \tau_k\); or creating an actor that operates in a dataspace of a certain communication type, \(\text{Spawn} \ \tau_k\). Type environments \(\Gamma\) associate variable names from patterns and fields with their types and facet names with the marker FacetName.

Figure 4.14 specifies the type judgment on facets: \(\Gamma \vdash \Pr \leadsto \text{Eff}\). It ensures that facet and field names are used appropriately while collecting certain communication operations as effect types. Auxiliary judgments apply to the different categories of syntax; section 4.3.1 provides the full 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 assertions \(A\), which also produces a description of the assertion types \(\tau_k\). The event descriptors \(D\) for each event handler are checked in the same manner, yielding a pattern type \(\tau_P\) for each expected input. 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 \(\text{bindings}^D\) metafunction; definition 55). Checking each body produces additional effects, \(\text{Eff}\). The product of the rule is the combination of outputs for each assertion template, as well as an interest for each external event handler, plus the effects from the event handler bodies.
\[ \tau \in \text{Type} = B \quad \tau_k \in \text{TmplTy} = * \]

\[ B \in \text{BaseTy} = \text{basic types: String, etc.} \]

\[ \Gamma \in \text{Env} = \cdot \]

\[ \text{Eff} \in \text{Effect} = \text{Output } \tau_k \quad \text{Input } \tau_k \quad \text{Spawn } \tau_k \]

\[ \text{Figure 4.13: Type Syntax} \]

\[ \Gamma \vdash \text{Pr} \Rightarrow \text{Eff} \]

\[ \Gamma \vdash \emptyset \Rightarrow \text{T-Null} \]

\[ \Gamma \vdash \text{Pr}_1 \Rightarrow \text{Eff}_1 \quad \Gamma \vdash \text{Pr}_2 \Rightarrow \text{Eff}_2 \quad \Gamma \vdash \text{Pr}_1; \text{Pr}_2 \Rightarrow \text{Eff}_1; \text{Eff}_2 \quad \text{T-Seq} \]

\[ \Gamma \vdash A \Rightarrow \tau_k \quad \Gamma \vdash \text{fn} : \text{FacetName} \]

\[ \Gamma \vdash \text{prune-up-to} \left( \text{fn}, \Gamma \right) \Rightarrow \text{Eff} \quad \text{T-Stop} \]

\[ \Gamma \vdash \text{spawn } \text{Pr} \Rightarrow \text{Eff} \quad \text{T-Spawn} \]

\[ \Gamma \vdash e : \tau \quad \Gamma, x \text{: Field} \tau \vdash \text{Pr} \Rightarrow \text{Eff} \quad \text{T-Field} \]

\[ \Gamma \vdash e : \tau \quad \Gamma(x) = \text{Field } \tau \quad \Gamma \vdash x := e \Rightarrow \text{Eff} \quad \text{T-Assign} \]

\[ \Gamma \vdash \text{let } x = e \text{ in } \text{Pr} \Rightarrow \text{Eff} \quad \text{T-Let} \]

\[ \text{Figure 4.14: Facet Typing} \]
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 (definition 56) 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 effects come directly from the continuation behavior.

T-FIELD, T-ASSIGN check field creation and assignment, respectively. Creating a field checks the type of the initial expression and associates the name with an appropriate type in the environment for the remaining statements. Field assignment looks up that type in the environment, making sure that the type of the assigned expression matches. Neither form results in any communication effects.

T-SPAWN concerns the creation of actors in the context of a dataspace with communication type $\tau_c$. The rule utilizes the prune function (definition 56) to remove identifiers associated with fields and facet names from the running actor. These names refer to local aspects of the actor doing the spawning, and have no meaning in the context of the spawned actor. Within this restricted context, checking the spawned actor’s behavior Pr yields a collection of communication effects. The type of expected inputs for the actor, $\tau_{\text{in}}$, is the union of each Input effect. Likewise, the potential outputs are the product of each Output effect. The input, output, and dataspace types must be in agreement in the same manner as the type judgment of section 3.2.3. That is, the inputs must include the potential outcomes of routing, as determined by the predict-routing metafunction (figure 3.7). Spawned actors—Spawn effects—must be consistent with the current dataspace communication type, according to actor subtyping (definition 27). The rule produces a single Spawn effect.

4.3.1 Auxiliary Definitions

The following definitions complete the type judgment for facet actor terms.

**Definition 50** ($\Gamma \vdash_D D \leadsto \tau_p \mid \cdot$). The judgment for event descriptions may relate an element of $D$ to either a pattern type $\tau_p$ or the empty sequence $\cdot$.

\[
\begin{align*}
\Gamma \vdash_D \text{start} & \leadsto \cdot & \text{D-START} \\
\Gamma \vdash_D \text{stop} & \leadsto \cdot & \text{D-STOP} \\
\Gamma \vdash_P P & \leadsto P & \text{D-ASSERTED} \\
\Gamma \vdash_D \text{asserted \ } P & \leadsto \tau_p & \text{D-ASSERTED} \\
\Gamma \vdash_D \text{retracted \ } P & \leadsto \tau_p & \text{D-ASSERTED}
\end{align*}
\]
Definition 51 \((\Gamma \vdash_A A \rightsquigarrow \bar{\tau}_k)\). The judgment for a union of assertions.

\[
\frac{\Gamma \vdash_A \emptyset \rightsquigarrow \cdot}{\text{A-\textsc{None}}} \quad \frac{\Gamma \vdash_k k : \tau_k \quad \Gamma \vdash_A A \rightsquigarrow \bar{\tau}_k}{\text{A-\textsc{Union}}}
\]

Definition 52 \((\Gamma \vdash_P P \rightsquigarrow \bar{\tau}_P)\). The judgment for patterns.

\[
\frac{\Gamma \vdash_P \star \rightsquigarrow \star}{\text{P-\textsc{Wildcard}}} \quad \frac{\Gamma \vdash_P x : \tau \rightsquigarrow x : \tau}{\text{P-\textsc{Bind}}} \quad \frac{\Gamma \vdash_P m(P) \rightsquigarrow m(\bar{\tau}_P)}{\text{P-\textsc{TUPLE}}} \quad \frac{\Gamma \vdash_P \text{observe} P \rightsquigarrow \text{observe} \bar{\tau}_P}{\text{P-\textsc{Observe}}}
\]

\[
\frac{\Gamma \vdash_e \text{ev} : \tau}{\text{P-\textsc{Expr}}} \quad \frac{\Gamma \vdash_e = \text{ev} \rightsquigarrow = \tau}{\text{P-\textsc{Observer}}}
\]

Definition 53 \((\Gamma \vdash_k k : \tau_k)\). The judgment for assertion templates.

\[
\frac{\Gamma \vdash_k \star : \star}{\text{K-\textsc{Wildcard}}} \quad \frac{\Gamma \vdash_k k : \tau_k}{\text{K-\textsc{Tuple}}} \quad \frac{\Gamma \vdash_k \text{observe} k : \text{observe} \bar{\tau}_k}{\text{K-\textsc{Observer}}} \quad \frac{\Gamma \vdash_k \text{ev} : \tau_k}{\text{K-\textsc{Expr}}}
\]

Definition 54 \((\Gamma \vdash_e e : \tau)\). The judgment for expressions.

\[
\frac{\Gamma \vdash_e b : B}{\text{E-\textsc{Base}}} \quad \frac{\Gamma(x) = \tau}{\text{E-\textsc{Var}}} \quad \frac{\Gamma(x) = \text{Field} \tau}{\text{E-\textsc{Field}}} \quad \frac{\Gamma \vdash_e ! x : \tau}{\text{E-\textsc{Ref}}} \quad \frac{\Gamma \vdash_e p(\bar{\tau}) : \tau'}{\text{E-\textsc{PrimOp}}} \quad \frac{\Gamma \vdash_e e : \tau}{\text{E-\textsc{Observe}}} \quad \frac{\Gamma \vdash_e m(\bar{\tau}) : m(\bar{\tau'})}{\text{E-\textsc{Tuple}}}
\]

The following metafunctions complete the definition of the type judgment.

Definition 55 \((\text{bindings}_{\text{D}})\). The \text{bindings}_{\text{D}} metafunction creates a type environment from the binding variables in an event description.

\[
\begin{align*}
\text{bindings}_{\text{D}} : & \quad \text{D} \rightarrow \Gamma \\
\text{bindings}_{\text{D}}(\text{start}) &= \cdot \\
\text{bindings}_{\text{D}}(\text{stop}) &= \cdot \\
\text{bindings}_{\text{D}}(\text{asserted} P) &= \text{bindings}_{\text{P}}(P) \\
\text{bindings}_{\text{D}}(\text{retracted} P) &= \text{bindings}_{\text{P}}(P)
\end{align*}
\]
It utilizes a helper function for patterns, $\text{bindings}_p$.

$$
\begin{align*}
\text{bindings}_p & : \mathcal{P} \rightarrow \Gamma \\
\text{bindings}_p(*) & = \cdot \\
\text{bindings}_p(x : \tau) & = x : \tau \\
\text{bindings}_p(m(\overrightarrow{P})) & = \text{bindings}_p(P) \\
\text{bindings}_p(\text{observe } P) & = \text{bindings}_p(P) \\
\text{bindings}_p(\_ = e) & = \cdot 
\end{align*}
$$

**Definition 56** (*prune*). 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.

$$
\begin{align*}
\text{prune} & : \Gamma \rightarrow \Gamma \\
\text{prune}(\cdot) & = \cdot \\
\text{prune}(\Gamma, \text{fn} : \text{FacetName}) & = \text{prune}(\Gamma) \\
\text{prune}(\Gamma, x : \tau) & = \begin{cases} 
\text{prune}(\Gamma) & \text{if } x = \text{Field } \tau' \\
\text{prune}(\Gamma), x : \tau & \text{otherwise} 
\end{cases}
\end{align*}
$$

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.

$$
\begin{align*}
\text{prune-up-to} & : \text{fn} \times \Gamma \rightarrow \Gamma \\
\text{prune-up-to}(\text{fn}, \cdot) & = \cdot \\
\text{prune-up-to}(\text{fn}, \Gamma, \text{fn}' : \text{FacetName}) & = \begin{cases} 
\Gamma & \text{if } \text{fn} = \text{fn}' \\
\text{prune-up-to}(\text{fn}, \Gamma) & \text{otherwise} 
\end{cases} \\
\text{prune-up-to}(\text{fn}, \Gamma, x : \tau) & = \text{prune-up-to}(\Gamma, \text{fn}), x : \tau
\end{align*}
$$

**Definition 57** (*pattern-sub*). The *pattern-sub* metafunction describes the type of interest that a pattern type $\tau_p$ gives rise to.

$$
\begin{align*}
\text{pattern-sub} & : \tau_p \rightarrow \tau_k \\
\text{pattern-sub}(\_*) & = \_ \\
\text{pattern-sub}(x : \tau) & = \_ \\
\text{pattern-sub}(m(\overrightarrow{\tau_p})) & = m(\text{pattern-sub}(\overrightarrow{\tau_p})) \\
\text{pattern-sub}(\text{observe } \tau_p) & = \text{observe pattern-sub}(\tau_p) \\
\text{pattern-sub}(\_ = \tau) & = \tau
\end{align*}
$$

### 4.3.2 Machine Typing

The statement of soundness for facet-based actors relies on two auxiliary notions. First, a machine state $M$ is inert, $\text{inert}(M)$, iff $M = \langle \text{FT}; \_ ; \; ; \pi; \sigma \rangle$. That is, it has neither
instructions to perform nor pending scripts to execute. Second, a designated set of transition labels \( l \) are dubbed *internal*. Internal labels correspond to the machine processing instructions and scripts. Thus, they consist of each \( \cdot \) and \( Fr \) label, but not \( ExtEvt \).

**Definition 58** (Store and Set Types). There are two extra notions of syntax for checking machine states. First, a store typing \( \sigma \tau \) maps store-bound variables to their types. Second, an assertion set typing \( \theta \) describes the different types of assertions known to the actor.

\[
\sigma \tau \in \text{StoreTy} = \{ x : \tau_k \ 
\theta \in \text{AssertionTys} = \bigcup \tau_k
\]

**Definition 59** (\( \vdash_M M \)). The type judgment for facet machines refers to the judgment for assigning types to assertion sets from definition 26.

\[
\begin{align*}
\vdash_M \sigma \tau & : \sigma \tau \\
\pi \vdash \theta & \quad \Gamma;\theta \vdash_{\text{FT}} FT \quad \Gamma \vdash \Gamma \quad \Gamma \vdash_{\text{PS}} PS
\end{align*}
\]

\( \text{T-MACHINE} \)

\[
\vdash_M \text{error} \quad \text{T-ERROR}
\]

**Definition 60** (\( \vdash_{\sigma} \sigma : \sigma \tau \)). The type judgment for stores.

\[
\sigma \tau = \{ x : \tau | x : \tau \in \sigma \} \quad \vdash_{\sigma} \sigma \tau
\]

\( \sigma \text{-SET} \)

**Definition 61** (*store-bindings*). This function creates a type environment from a store type.

\[
\begin{align*}
\text{store-bindings} & : \sigma \tau \rightarrow \Gamma \\
\text{store-bindings}(\emptyset) & = \cdot \\
\text{store-bindings}(\sigma \tau \cup x : \tau) & = \text{store-bindings}(\sigma \tau), x : \tau
\end{align*}
\]

**Definition 62** (\( \Gamma;\theta \vdash_{\text{FT}} FT \)). The type judgment for facet trees. It refers to the project-safe relation from definition 25.

\[
\begin{align*}
\Gamma;\theta & \vdash_{\text{FT}} \epsilon \quad \text{FT-EMPTY} \\
\Gamma;\theta & \vdash_{\text{FT}} D \quad \text{FT-TREE}
\end{align*}
\]
**Definition 63** ($\Gamma \vdash_I$). The type judgment for machine instructions.

\[
\Gamma \vdash A \rightsquigarrow \ldots \rightsquigarrow \Gamma \vdash D \rightsquigarrow \ldots \\
\text{fid} = (\text{fn}^\prime) \quad \Gamma, \text{bindings}_D(D), \text{fn}^\prime : \text{FacetName}, \text{fn} : \text{FacetName} \vdash \text{Pr} \rightsquigarrow \text{I-START} \\
\Gamma \vdash \text{start fn } A (D \text{ Pr}) @ \text{fid} \\
\Gamma \vdash \text{stop fn} \quad \text{I-STOP} \\
\text{prune}(\Gamma) \vdash \text{Pr} \rightsquigarrow \text{I-SPAWN} \\
\Gamma \vdash \text{spawn Pr} \quad \text{PS-SCRIPT} \\
\]

**Definition 64** ($\Gamma \vdash_{PS} \text{PS}$). The type judgment for pending scripts.

\[
\Gamma, \text{fn} : \text{FacetName} \vdash \text{Pr} \rightsquigarrow \Gamma \vdash_{PS} (\langle \text{fn} \rangle, \text{Pr}) \\
\]

### 4.4 Properties

The statement of soundness utilizes a notion of the project-safe relation (definition 25) lifted to facet machines.

**Definition 65** (project-safe(ExtEvt, M)). A machine configuration $M = \langle FT; T; \text{PS}; \pi; \sigma \rangle$ is safe for projection with respect to an event ExtEvt if $\text{ExtEvt} = \pi^+ / \pi^-$ implies $\pi^+ \cup \pi^- \models \tau$ and, for each pattern type $\tau_P$ in FT, project-safe($\tau, \tau_P$).

**Theorem 66** (Soundness). If $\vdash_M M$, project-safe(ExtEvt, M), and $M \xrightarrow{\text{ExtEvt}} M'$ then either:

- $M' \xrightarrow{\text{inert}} M''$ and inert($M''$); or
- $M' \xrightarrow{\text{error}}$

**Proof** By the standard progress (lemma 67) and preservation (lemma 68) properties. 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. □.

**Interpretation.** 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).

**Lemma 67** (Progress). If $\vdash_M M$ and $M \neq \text{error}$ then either inert($M$) or there exists $M'$ such that $M \xrightarrow{\cdot} M'$.

**Proof** By case analysis on the machine state M. □
Lemma 68 (Preservation). If

- \( \vdash_M M \)
- \( M \overset{l}{\rightarrow} M' \)
- \( l = \text{ExtEvt} \implies \text{project-safe}(\text{ExtEvt}, M) \)

then

- \( \vdash_M M' \).

Proof The metafunctions \textit{dispatch} and \textit{p-e} perform the most significant transformations on the machine state. Lemma 69 shows that \textit{dispatch} preserves type structure appropriately while lemma 70 does so for \textit{p-e}. \(\square\)

Lemma 69 (Dispatch). If

- \( \Gamma; \theta \vdash_{\text{FT}} \text{FT} \)
- \( \text{dispatch}(\text{FT}, \text{Evt}, \pi, \pi', \sigma, \langle \text{fn} \rangle) = \overset{\rightarrow}{\text{PS}} \)
- \( \vdash_\sigma \sigma : \sigma_\tau \)
- \( \Gamma = \text{store-bindings}(\sigma_\tau), \text{fn} : \text{FacetName} \)
- \( \pi \cup \pi' \models \theta \)
- \( \text{Evt} = \pi^+ / \pi^- \implies \pi^+ \cup \pi^- \models \theta \)

then

- \( \Gamma \vdash_{\text{PS}} \overset{\rightarrow}{\text{PS}} \).

Proof The primary requirements to show \( \Gamma \vdash_{\text{PS}} \text{PS} \) are that \( \Gamma \) binds the field names in \( \text{PS} \), the fid for each pending script binds its facet names, and that event matching/projection succeeds (\textit{dispatch} is defined) and preserves types.

For the first requirement, it suffices to observe that \textit{dispatch} does not create any fields, thus the field names referred to by the produced pending scripts were also present in the initial facet tree \( \text{FT} \), so must be accounted for by \textit{store-bindings}(\sigma_\tau).

For facet names, \textit{dispatch} accumulates the names of facets on the path through the tree. The contents of that accumulator are used by \textit{dispatch1} when instantiating pending scripts, so it includes every facet that the bodies of event handlers may refer to.

Finally, the success of event matching and projection is a consequence of ensuring that the assertions sets involved in the \textit{dispatch} have an appropriate type. The judgment \( \Gamma; \theta \vdash_{\text{FT}} \text{FT} \) checks that the set types are \textit{project-safe} with respect to each event handler in the actor, ensuring the success of projection via lemma 32. \(\square\)
Lemma 70 (Partial Evaluation). If

- $\Gamma \vdash Pr \sim_\_\_
- \vdash_{\sigma} \sigma : \sigma_T
- \Gamma = \text{store-bindings}(\sigma_T), \text{fn : FacetName}
- p-e(Pr, \sigma, (\text{fn})) = \text{\_}, \sigma'

then

- $\Gamma' \vdash_{I} I$
- $\vdash_{\sigma} \sigma' : \sigma'_T$
- $\Gamma' = \text{store-bindings}(\sigma'_T)$

Proof Most of the cases are immediate via induction on the type derivation. The interesting behavior involves starting and stopping facets.

A statement starting a facet, start fn A (D Pr), gives rise to a located start instruction. For the instruction to be well-typed, the location fid must include all free facet names in the statement. The fid supplied to p-e comes via dispatch, and as lemma 69 shows, describes the facets from the root of the facet tree to the particular facet. The p-e function passes along the fid unchanged, with the only exception being the case of a stop statement, described below.

Likewise, a stop statement, stop fn Pr, gives rise to a stop instruction. Typing a stop instruction is straightforward. The tricky bit is that the continuation behavior is p-e’d in the context of a manipulated fid. Crucially, the surgery performed by p-e on the fid context mirrors the definition of the prune-up-to function (definition 56) employed by rule T-Stop. □.

Finally, initial machine states are well-typed.

Lemma 71 (Boot). If

- $\cdot \vdash Pr \sim_\_\_$
- $\text{boot}_{Pr}(Pr, \emptyset) = M$

then

- $\vdash_{M} M$

Proof From the definition of boot_{Pr}, it must be that $\overrightarrow{I} = \cdot$ and $\pi = \emptyset$, so $\theta = \bigcup \cdot$. Thus, what needs to be shown is that the machine’s initial facet tree, FT, and pending scripts, PS, are well-typed under the bindings of the store. The first case of the definition of boot_{Pr} handles fields; the other cases simply pass along the store argument before
using it in the initial machine state. The function extends the store for each bound field, and ensures each reference to the field uses the store-bound name via substitution. Therefore, if the initial store passed to boot, is empty, every field reference is accounted for. Via this field coverage and lemma 69, the facet tree and pending scripts are well typed. □

Lemma 72 (Linking). interp_M ∈ BehFun (figure 2.8).

Proof Taking State = Machine, interp_M (definition 48) covers all possible inputs. □

Lemma 73 (Safe Event Dispatch). If the dataspace dispatches a patch Δ to an actor with state M, then project-safe(Δ, M).

Proof As observed by lemma 36, the types of assertions in patches are consistent with the types of assertions produced by actors. Note that it cannot be the case that M = error, because in order to receive an event, the actor must have an active subscription. Theorems 28 and 34 imply that predict-routing, invoked by rule T-SPAWN, ensures the safety of projection. □

Theorem 74 (System Soundness). If for a fixed communication type τ_c, \( \vdash \text{spawn Pr} \leadsto \text{Spawn } \tau_c \), then boot_{DS}(\text{spawn Pr}) = DS with either

- DS \( \rightarrow^{*}_{DS} DS' \) and inert_{DS}(DS'); or
- for all DS', DS \( \rightarrow^{*}_{DS} DS' \) implies there exits DS'' such that DS' \( \rightarrow^{*}_{DS} \Sigma'' \)

Proof By theorem 10 and relating the input, output, and spawn effects of the type judgment to the function types from chapter 3. □

4.5 RELATED WORK

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 [106], the Mobile Ambient Calculus [15], the join calculus [36], and tuplespaces [16, 75], as well as various actor systems and process calculi. Hence, this section focuses on linguistic features for organizing the code of individual actors.

State-Machine Actors. Actor systems such as Akka [3] and Erlang/OTP [7] 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 4.1.

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. Demultiplexing is automated only to the extent that different types of messages are in a one-to-one correspondence with machine states. 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 [74] shares many similarities with the dataspace model. Fact spaces build on TOTAM [46, 93, 94], 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 records (implicitly) the dependencies between facts and invokes application actions specified by logic rules.

The implementation of the fact space model, dubbed CRIME [73], integrates the fact spaces model with the AmbientTalk language [102, 103]. AmbientTalk is an instance of the Actor model [2, 26] in the mold of E [71]. 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 [21]. 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 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. Researchers on Active Objects recognized the need for concurrency within an Actor or Actor-like entity [10, 90, 91]. The situation
is similar to AmbientTalk and E. The difference is that multiple message dispatches may be active within a single active object. This allows for splitting control and state among numerous objects, resembling the way facets and fields individually contribute to the actor’s overall behavior. Then again, these systems lack the hierarchical structure of facets, especially with respect to orderly startup and termination, as well as the dataflow that links local and shared state.

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 [11]. Each active object (node in the communication graph) may implement its behavior using any number of such passive objects following traditional object-oriented design [39].

Dataspaces and faceted actors possess the same graph-like communication structure. Each actor (node) is connected to each other actor that it communicates with (edge) via assertion(s). 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.

**Sparrow.** The Sparrow DSL [8] 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. Reactions lack the other features of the 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 utilized by facets is most similar to the dependency tracking approach to object-oriented reactive programming described by Salvaneschi and Mezini [86, section 2.3], and was in fact directly inspired by the dependency tracking of JavaScript frameworks such as Knockout [87] and Meteor.\footnote{https://docs.meteor.com/api/tracker.html}
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, check it statically (as much as possible).

This chapter builds such a mechanism using the “types-as-processes” technique [17, 60] combined with an off-the-shelf model checker. That is, the effect types from section 4.3 are extended to capture the communication behavior of facets, and those behavioral descriptions serve as a basis for model checking the program with respect to specifications stated in a temporal logic. Section 5.1 gives an overview of the approach, while sections 5.2 and 5.3 provide the formal semantics and key meta-theorems. Section 5.4 describes the implementation of the checking tool and section 5.5 reports the evaluation. Section 5.6 concludes the chapter with a discussion of related work.

5.1 OVERVIEW

The language of facets comes with a structural type system and a behavioral one. The first is somewhat conventional (sections 4.3 and 5.1.1). The second enables programmers to specify an actor’s dynamic behavior (section 5.1.2) and perform some static checking (section 5.1.4). The bridge between the two is an effect-type system (section 5.1.3).

5.1.1 Basic Types

The starting point is the structural type system like that of sections 3.1 and 4.3. Base types describe data such as Int, Bool, etc. A struct-type definition such as (struct light ([on? : Bool])) introduces the type constructor LightT. Thus, (light #true) has type (LightT Bool), or just Light.

5.1.2 Specification

Like for structural properties of routing assertions, programmers need a language for behavioral—that is, temporal—properties, too. A form of linear temporal logic (LTL) [82] suggests itself here. LTL is a common specification language in the setting of (concurrent) system verification [56, 84, 96] and is arguably more intuitive than other options [104]. 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.

The basic LTL predicates are just the types that describe assertions in the dataspace.
For example, the formula \((\text{LightT Bool})\) holds at any moment iff some actor has
deposited an assertion \((\text{light } #\text{true})\) or \((\text{light } #\text{false})\) in the dataspace. Specifi-
cations may also use negation \((\text{Not})\), and \((\text{Not } (\text{LightT Bool}))\) means no assertion of this
structure type is in the dataspace. Other connectives are conjunction \((\text{And})\), disjunction
\((\text{Or})\), and logical implication \((\text{Implies})\).

With temporal connectives, a programmer can specify the evolution of dataspace
programs. The proposition \((\text{Always } (\text{LightT Bool}))\) holds if the dataspace contains
a light assertion at every moment during execution, regardless of which actor has
deposited it there. The property \((\text{Eventually } (\text{LightT Bool}))\) describes an execution
state with an assertion matching \((\text{LightT Bool})\) in the dataspace or with the execution
arriving at such a state within a finite number of steps. Meanwhile,

\[(\text{Until } (\text{LightT Bool}) (\text{Not } (\text{LightSwitchStateT Symbol Bool}))))\]
says that eventually there will not be a light-switch-state assertion in the dataspace,
but until then there is a light assertion.\(^1\)

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 formula
\((\text{Eventually } (\text{LightT Bool}))\) if, at some point during execution, an actor makes a
matching light assertion. In particular, any dataspace program that includes the
light-conversation actor from the preceding section satisfies this formula, because it
makes a light assertion as it boots.

Consider another proposition:

\[(\text{Always })
(\text{Implies } (\text{LightT Bool})
(\text{Until } (\text{LightT Bool}) (\text{Not } \text{WallSwitchOn})))\]

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.

Take the dataspace program that runs these two actors:

\[
\begin{array}{ll}
\text{actor 1} & \text{actor 2} \\
(\text{start light-conversation}) & (\text{start wall-switch}) \\
(\text{field state ON}) & (\text{assert wall-switch-on}) \\
(\text{assert light } (! \text{ state})) & (\text{on asserted light } _) \\
(\text{on (retracted wall-switch-on)}) & (\text{stop wall-switch}) \\
(:= \text{ light OFF})) & \\
\end{array}
\]

\(^1\) This is the “strong” version of the Until connective, requiring that \((\text{LightSwitchStateT Symbol Bool})\)
eventually be true.
The table below shows the evolution of the dataspace. It shows which assertions are in the dataspace for the first four time steps, starting with the boot state.

<table>
<thead>
<tr>
<th>t = assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

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 matches 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. 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.

### 5.1.3 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 [69]. In the setting of facet-oriented actors, the to-be-observed effects are caused by actors overall, facets, and endpoints. Much like the type system from section 4.3 recorded input and output communication types of actors, each corresponding term-level construct comes with a type-level construct for describing the type of its effects. See figure 5.1 for the actual notation.

The following listing illustrates the synthesis of effect types with the code for spawning the simplistic light-actor from section 4.1 (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
Also, $x:T$ and $\_\_\_$ have types $T$ and $\_\_$, respectively.

Figure 5.1: Effect Types

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:

\[
\begin{align*}
\text{(spawn)} & \quad \text{(Spawn)} \\
\text{(start-facet light-conversation)} & \quad \text{(StartFacet light-conversation)} \\
\text{\hspace{1cm} (field [light-state ON])} & \quad \text{\hspace{1cm} (Assert (LightT Bool))} \\
\text{\hspace{1cm} (assert (light (! light-state)))} & \quad \text{\hspace{1cm} (On (Asserted (InRoomT Int)))} \\
\text{\hspace{1cm} (on (asserted (in-room 0))}} & \quad \text{\hspace{1cm} \hspace{1cm} (On (Retracted (InRoomT Int))))} \\
\text{\hspace{1cm} \hspace{1cm} (:= light-state OFF))} & \quad \text{\hspace{1cm} \hspace{1cm} \hspace{1cm}} \\
\text{\hspace{1cm} (on (retracted (in-room 0))}} & \quad \text{\hspace{1cm} \hspace{1cm} \hspace{1cm}} \\
\text{\hspace{1cm} \hspace{1cm} (:= light-state ON)))} & \quad \text{\hspace{1cm} \hspace{1cm} \hspace{1cm}}
\end{align*}
\]

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 values. Taking the behavioral type of each actor in a program—say, the smart home program—gives 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.

5.1.4 Checking Specifications Mechanically

The transition of possible interactions from concrete value assertions to types enables specification checking. At a 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 section 5.2. Section 5.3 verifies why this approach works. The implementation is sketched in section 5.4. Section 5.5 discusses the evaluation. The following subsection illustrates the working of the model checker with an example.

5.1.5 Revisiting the Smart-Home Example

Revisiting the smart home example from section 4.1 demonstrates the power of the specification language. The code presented in figure 4.3 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:

\[
\begin{align*}
&\text{(Always (And (Implies RoomOccupied}
&\text{ (Implies WallSwitchOn (Eventually LightOn)))}
&\text{(Implies RoomEmpty}
&\text{ (Eventually (Not LightOn)))))}
\end{align*}
\]

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, the implementation of LTL checking reports the error and provides a counterexample:

```
... Process Sensor ASSERTS RoomEmpty ...
... Process Switch ASSERTS WallSwitchOn ...
... Process Light ASSERTS LightOff
Violation: ...
```

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.

The trace allows for identifying 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-conversation facet in figure 4.3 reacts only to the retraction of (in-room room 0) to determine that the room is occupied. If the room is occupied when the light regains power, no such retraction will take place. Fixing this problem entails amending the facet to query the state of in-room assertions on startup. The boxed code in figure 5.2 is all that is needed to make the actor satisfy its LTL specification.
(define (spawn-light wall-switch-id)
  (define my-id (generate-unique-id "light"))
  (spawn
    (start-facet light-conversation
      (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))
          (= on? (not (zero? n))))
        (stop init))))))

Figure 5.2: Implementation of the light-actor, fixed (see figure 4.3)

IMPLEMENTATION NOTE. This example utilizes an experimental feature of the implementation that enables reasoning about true/false and zero/nonzero values in assertions, described in section 5.4.4. The basis of this reasoning is an extension to the type-and-effect system to include information about field creation, reads, and writes, in addition to facets and endpoints. Using information about field types, it can perform typestate-style reasoning on the influence of field updates on the actor’s assertions.

5.2 TYPE LEVEL SEMANTICS

This section defines the communication behavior of type-level facet descriptions and the formal meaning of LTL propositions concerning dataspaces.

Figure 5.3 defines the syntax of types. Effect types $T$ directly reflect the core facet syntax (figure 4.7), including facet start/stop and spawn operations of Pr. Additional aspects of effect types correspond directly to the term level: $D_T$ to $D$, $A_\tau$ to $A$, $\tau_k$ to $k$, and $\tau_P$ to $P$. Types $\tau$ summarize the result of expressions, with base types $B$ describing primitive values. Type environments $\Gamma$ associate variable names from patterns and fields with their types and facet names with the marker FacetName.

Figure 5.4 specifies the core type judgment on facets: $\Gamma \vdash_{Pr} Pr : T$. Like the judgment defined in section 4.3, 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; section 5.2 provides the full definitions.

The typing rules perform many of the same checks as the structural type judgment on facets (figure 4.14). The explanations below focus on the differences:

T-START assigns types to a facet-creation instruction. It appeals to an auxiliary judgment for checking the assertions $A$, which also produces a description of the
T ∈ FacetTy = Start fn A \tau (D_T \mapsto T) \\
| Stop fn T \\
| Spawn T \\
| T; T \\
| \emptyset

\tau ∈ Type = \text{see figure 4.13}

\tau_p ∈ PatTy = \text{see figure 4.13}

D_T ∈ EvtDscTy = asserted \tau_p \\
| retracted \tau_p \\
| start \\
| stop

A_\tau ∈ AsrtTy = \emptyset \\
| \tau_k \cup A_\tau

\text{see figure 4.13}

\text{Figure 5.3: Type Syntax}

\[\begin{align*}
\Gamma \vdash Pr : T \\
\Gamma \vdash Pr : T & \quad (\text{T-NULL}) \\
\Gamma \vdash A : A_\tau \quad \Gamma \vdash D : D_T \\
\Gamma, \text{fn : FacetName, bindings}_D(D) \vdash Pr : T & \quad (\text{T-START}) \\
\Gamma \vdash Pr : T & \quad (\text{T-SEQ}) \\
\Gamma \vdash Pr : T & \quad (\text{T-STOP}) \\
\Gamma \vdash Pr : T & \quad (\text{T-SPAWN}) \\
\Gamma \vdash x : \tau \quad \Gamma, x : \text{Field} \tau \vdash Pr : T & \quad (\text{T-FIELD}) \\
\Gamma \vdash x : \tau \quad \Gamma(x) = \text{Field} \tau & \quad (\text{T-ASSIGN}) \\
\Gamma \vdash x := e : \emptyset \quad \Gamma \vdash e : \tau & \quad (\text{T-LET})
\end{align*}\]

\text{Figure 5.4: Facet Typing}
assertion types $A_T$. The event descriptors $D$ for each event handler are checked in the same manner, yielding a type $D_T$ for each. The final aspect to check is the body $Pr$ of each event handler. 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. The resulting effect type is to $Stop$ the same facet with a continuation behavior.

$T-Field$, $T-Assign$ check field creation and assignment, respectively. 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 null effect.

$T-Spawn$ concerns the creation of actors. The rule utilizes the $prune$ function (definition 56) to remove identifiers associated with fields and facet names from the running actor, because these names are local. The spawned actor’s behavior $Pr$ must have the (sole) effect of $Start$-ing a facet. The effect type of the $spawn$ statement is a $Spawn$ with a type description of the initial facet.

**Auxiliary Type Judgments**

The following definitions complete the type judgment for facet actor terms.

**Definition 75** ($\Gamma \vdash_A A : A_T$). The judgment for a union of assertions.

$$
\begin{align*}
\Gamma & \vdash_A \emptyset : \emptyset & \text{A-None} \\
\Gamma & \vdash_A k : \tau_k & \Gamma \vdash_A A : A_T & \Gamma \vdash A \cup k : \tau_k \cup A_T & \text{A-Union}
\end{align*}
$$

**Definition 76** ($\Gamma \vdash_D D : D_T$). The judgment for event descriptions.

$$
\begin{align*}
\Gamma & \vdash_D \text{start} : \text{start} & \text{D-Start} \\
\Gamma & \vdash_D \text{stop} : \text{stop} & \text{D-Stop} \\
\Gamma & \vdash_D \text{asserted} P : \text{asserted} \tau_P & \text{D-Asserted} \\
\Gamma & \vdash_D \text{retracted} P : \text{retracted} \tau_P & \text{D-Asserted}
\end{align*}
$$
Definition 77 ($\Gamma \vdash_{P} P : \tau_{P}$). The judgment for patterns.

\[
\begin{align*}
\Gamma \vdash_{P} * : * & \quad \text{P-WILDCARD} \\
\Gamma \vdash_{P} x : \tau & \quad \text{P-BIND} \\
\Gamma \vdash_{P} m(\overrightarrow{P}) : m(\overrightarrow{\tau_{P}}) & \quad \text{P-TUPLE} \\
\Gamma \vdash_{P} \text{observe } P : \text{observe } \tau_{P} & \quad \text{P-OBSERVE} \\
\Gamma \vdash_{e} e : \tau & \quad \text{P-EXPR}
\end{align*}
\]

5.2.1 Machine Types and Type-level Machines

Figure 5.5 defines the syntax for describing the states of a type-level facet machine. In brief, the syntax of machine types reflects the syntax of machines (figure 4.8) in the same manner that effect types $T$ (figure 5.3) reflect the syntax of statements $Pr$ (figure 4.7). Figure 5.6 defines the transition relation for a type-level facet machine. The definitions of the metafunctions are mutatis mutandis those from section 4.2.3; their definitions are omitted.

Definition 78 ($\vdash_{M} M : M_{T}$). The judgment $\vdash_{M} M : M_{T}$ relates a machine $M$ to a machine type $M_{T}$. A machine type $\langle FT_{T}; IT_{T}; PS_{T}; \pi_{T}; \sigma_{T} \rangle$ includes a facet tree type $FT_{T}$, sequence of instruction types ($IT_{T}$), sequence of pending script types ($PS_{T}$), set of assertion types ($\pi_{T}$), and store type ($\sigma_{T}$).

\[
\begin{align*}
\text{store-bindings}(\sigma_{T}) &= \Gamma \\
\Gamma \vdash_{FT} FT : FT_{T} \quad \Gamma \vdash_{I} I : IT_{T} \quad \Gamma \vdash_{PS} PS : PS_{T} \\
\vdash_{\pi} \pi : \pi_{T} \quad \vdash_{\sigma} \sigma : \sigma_{T} \quad \pi = \theta \quad \Gamma, \theta \vdash_{FT} FT \\
\vdash_{M} \langle FT_{T}; IT_{T}; PS_{T}; \pi_{T}; \sigma_{T} \rangle & \quad \text{M-MACHINE}
\end{align*}
\]

Definition 79 ($\Gamma \vdash_{FT} FT : FT_{T}$). The type judgment for facet trees.

\[
\begin{align*}
\Gamma \vdash_{FT} e : e & \quad \text{FT-EMPTY} \\
\Gamma \vdash_{A} A : A_{T} \\
\Gamma, fn : \text{FacetName}, bindings_{D} \vdash_{Pr} Pr : T \\
\Gamma, fn : \text{FacetName} \vdash_{FT} FT : FT_{T} \\
\Gamma \vdash_{FT} fn[A_{T}(\overrightarrow{D_{T}})].FT_{T} : fn[A_{T}(\overrightarrow{D_{T}})].FT_{T} & \quad \text{FT-TREE}
\end{align*}
\]
\[ M_T \in \text{TyMachine} = (F_T; I_T; P^\rightarrow_T; \pi_T; \sigma_T) \]

\[ FT_T \in \text{FctTreeTy} = \epsilon \]
| fn[A_T(D_T T)] \cdot F_T \]

\[ CT \in \text{TyContext} = \square \]
| fid[A_T(D_T T)] \cdot F_T \cdot C_T \cdot F_T \]

\[ PST \in \text{PScriptTy} = (fid, T) \]

\[ I_T \in \text{InstrTy} = \text{start fn } A_T(D_T T) @ fid \]
| stop fn \]
| spawn T \]

\[ \delta_T \in \text{TyInterp} = p \times T \rightarrow T \]

\[ \gamma_T \in \text{TySub} = x \overset{\text{fn}}{\rightarrow} T \]

\[ \tau_R \in \text{SpecType} = \{ B \mid == R \} \]
| observe \tau_R \]
| m(\tau_R) \]

\[ R \in \text{Refine} = x | b \]

\[ \pi_T \in \text{ASetTy} = \mathcal{P}(\tau_R) \]

\[ \Delta_T \in \text{PatchTy} = \pi_T^+ / \pi_T^- \]
| \text{where } \pi_T^+ \cap \pi_T^- = \emptyset \]

\[ \text{ExtEvt}_T \in \text{ExtEventTy} = \Delta_T \]

\[ \text{Evt}_T \in \text{EventTy} = \text{ExtEvt}_T \]
| \text{start} \]
| \text{stop} \]

\[ I_T \in \text{LabelTy} = \cdot \]
| \text{ExtEvt}_T \]
| T \]

Figure 5.5: Evaluation Syntax for Type Machines
\[
\begin{align*}
(\text{FT}_T; \vdash; \pi_T; \sigma_T) & \xrightarrow{\text{FT}_T} (\pi_T; \sigma_T) & \text{(injectT)} \\
\text{where} & & \\
\pi_T' & = \pi_T \oplus T \text{ ExtEvt}_T \\
\text{PS}_T & = \text{dispatch}_T(\text{FT}_T, \text{ExtEvt}_T, \pi_T, \sigma_T, \langle \cdot \rangle)
\end{align*}
\]

\[
\begin{align*}
(\text{FT}_T; \langle \text{fid}, T \rangle \cdot \text{PS}_T; \pi_T; \sigma_T) & \xrightarrow{\text{M}_T} (\pi_T; \sigma_T) & \text{(transferT)} \\
\text{where} & & \\
\text{M}_T & = (\text{FT}_T; \text{I}_T; \text{PS}_T; \pi_T; \sigma_T') \\
\text{I}_T, \sigma_T' & = p_{-T}(T, \sigma_T, \text{fid})
\end{align*}
\]

\[
\begin{align*}
(\text{FT}_T; \text{start fn A}_T (D_T \cdot \text{I}_T) @ \text{fid} \cdot \text{PS}_T; \pi_T; \sigma_T) & \xrightarrow{\text{M}_T} (\text{FT}_T; \text{I}_T; \text{PS}_T; \pi_T; \sigma_T) & \text{(startT)} \\
\text{where} & & \\
\text{fn}_{\text{new}} & \text{ fresh in } \text{FT}_T \\
\text{FT}_{T_{\text{new}}} & = \text{fn}_{\text{new}}[A_T (D_T [\text{fn} \mapsto \text{fn}_{\text{new}}]), \epsilon] \\
\text{PS}_{T_{\text{start}}} & = \text{dispatch}_T(\text{FT}_{T_{\text{new}}}, \text{start}, \emptyset, \emptyset, \sigma_T, \text{fid}) \\
\text{PS}_{T_{\text{boot}}} & = \text{dispatch}_T(\text{FT}_{T_{\text{new}}}, \pi_T, \emptyset, \emptyset, \sigma_T, \text{fid}) \\
\text{FT}'_T & = \begin{cases} 
\text{C}_T[\text{FT}_{T_{\text{new}}}] & \text{if locate}_T(\text{FT}_T, \text{fid}) = \text{C}_T \\
\text{FT}_T & \text{otherwise}
\end{cases} \\
\text{PS}_{T_{\text{stop}}} & = \begin{cases} 
\text{C}_T[\text{FT}_{T_{\text{new}}} - \text{if locate}_T(\text{FT}_T, \text{fid}) \text{ defined} \\
\text{dispatch}_T(\text{FT}_{T_{\text{new}}}, \text{stop}, \pi_T, \sigma_T, \text{fid}) & \text{otherwise}
\end{cases} \\
\text{PS}_{T_{\text{next}}} & = \text{PS}_T \cdot \text{PS}_{T_{\text{stop}}} \cdot \text{PS}_{T_{\text{boot}}} \cdot \text{PS}_{T_{\text{stop}}} \\
\end{align*}
\]

\[
\begin{align*}
(\text{FT}_T; \text{stop fn A}_T (D_T \cdot \text{I}_T) @ \text{fid} \cdot \text{PS}_T; \pi_T; \sigma_T) & \xrightarrow{\text{M}_T} (\text{FT}_T; \text{I}_T; \text{PS}_T; \pi_T; \sigma_T) & \text{(stopT)} \\
\text{where} & & \\
\text{FT}_T & = \begin{cases} 
\text{C}_T[\epsilon] & \text{if } \text{FT}_T = \text{C}_T[\text{fn} \mapsto \text{FT}_T] \\
\text{FT}_T & \text{otherwise}
\end{cases} \\
\text{fid} & = \text{facet-context}_T(\text{C}_T) \\
\text{PS}_{T_{\text{stop}}} & = \begin{cases} 
\text{dispatch}_T(\text{fn} \mapsto \text{FT}_T, \text{stop}, \pi_T, \sigma_T, \text{fid}) & \text{if } \text{FT}_T = \text{C}_T[\text{fn} \mapsto \text{FT}_T] \\
\text{otherwise}
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
(\text{FT}_T; \text{spawn T} \cdot \text{I}_T; \text{PS}_T; \pi_T; \sigma_T) & \xrightarrow{T} (\text{FT}_T; \text{I}_T; \text{PS}_T; \pi_T; \sigma_T) & \text{(spawnT)}
\end{align*}
\]

Figure 5.6: Type Machine Transitions
Definition 80 ($\Gamma \vdash I : I_T$). The type judgment for machine instructions.

\[
\begin{align*}
\Gamma \vdash_A A : A_T \\
\Gamma \vdash_D D : D_T \\
\Gamma, \text{bindings}_D(D) \vdash_{Pr} Pr : T
\end{align*}
\]

\[
\Gamma \vdash_{\text{I-START}} \text{start fn } A (D \ Pr) @ \text{fid} : \text{start fn } A_T (D_T \ T) @ \text{fid}
\]

\[
\Gamma \vdash_{\text{I-STOP}} \text{stop fn} : \text{stop fn}
\]

\[
\Gamma \vdash_{\text{I-SPAWN}} \text{prune(}\Gamma\text{)} : \text{prune Pr} : T
\]

Definition 81 ($\Gamma \vdash_{\text{PS}} PS : \text{PS}_T$). The type judgment for pending scripts.

\[
\begin{align*}
\Gamma, \text{fn} : \text{FacetName} \vdash_{Pr} Pr : T \\
\Gamma \vdash_{\text{PS-SCRIPT}} \text{PS} (\langle \text{fn} \rangle, Pr) : \langle \text{fn} \rangle, T
\end{align*}
\]

Definition 82 ($\vdash_{\pi} \pi_T : \pi_T$). The type judgment for assertion sets.

\[
\begin{align*}
\pi_T = \{ \tau_R | c \in \pi_T, \vdash_{\tau_R} c : \tau \}
\end{align*}
\]

\[
\vdash_{\pi-SET} \pi : \pi_T
\]

Definition 83 ($\vdash_{\tau_R} c : \tau_R$). The type judgment for assertions.

\[
\vdash_{\tau_R} b : \{ B \mid \text{eq} b \} \quad \vdash_{\tau_R} \tau_R \tau
\]

\[
\vdash_{\tau_R} c : \tau_R \quad \vdash_{\tau_R} m(c) : m(\tau_R)
\]

\[
\vdash_{\tau_R} \text{observe } c : \text{observe } \tau_R
\]

Definition 84 ($\vdash_A \Delta : \Delta_T$). The type judgment for patches.

\[
\vdash_{\pi} \pi_T^+ : \pi_T^- \\
\vdash_A \Delta : \pi_T^+/\pi_T^-
\]

\[
\vdash_A \Delta : \Delta_T
\]

\[
\vdash_{\text{EVT-PATCH}} \Delta_T \quad \vdash_{\text{EVT-START}} \text{start} : \text{start}
\]

Definition 85 ($\vdash_{\text{Evt}} \text{Evt} : \text{Evt}_T$). The type judgment for events.

\[
\vdash_{\text{Evt}} \Delta : \Delta_T
\]

\[
\vdash_{\text{Evt}} \text{stop} : \text{stop}
\]
Definition 86 (\(\vdash_{\gamma} \gamma : \gamma_\tau\)). The type judgment for substitutions.

\[
\gamma_\tau = \{ x \mapsto \tau \mid x \mapsto v \in \gamma, \vdash_e v : \tau \} \\
\vdash_{\gamma} \gamma : \gamma_\tau
\]

Definition 87 (\(\vdash_l l : l_\tau\)). The type judgment for transition labels.

\[
\vdash_l \cdot : \cdot \hspace{1cm} \vdash_{\text{Evt}} \text{Evt} : \text{Evt}_\tau \hspace{1cm} \vdash_{\text{Pr}} \text{Pr} : \text{T} \hspace{1cm} \vdash_{\text{L-Spawn}}
\]

5.2.2 Type-level Dataspaces

The transition system for machine states operates on machine types with only minor syntactic adjustments. The phrase type machine refers to a machine state type in the context of the subject of a transition system. Section 5.3 makes use of this notion for behavioral analysis of actors. Transitions on type machines are well-defined.

Theorem 88 (Well-definedness). Either \(M_T \xrightarrow{t} M'_T\) or \(\text{inert}(M_T)\).

A type-level dataspace \(DS_T\) has the same semantics as those from section 2.3, with two modifications. First, assertions range over types \(\tau\). Second, actor behaviors are defined in terms of facet machine types, defined above, with related metafunctions such as \(\text{interp}_M\) lifted to the type level, mutatis mutandis.

5.3 Meta-Theorems

A semantics of facets, facet types, and dataspaces provides the basis for verifying the predictive nature of the behavioral types. Specifically, the semantics should show that if a program’s behavioral types satisfy a specification, the program itself does, too. The start is a model-appropriate definition of LTL specifications.

Definition 89 (LTL). \(\psi \in \text{LTL} = \tau \mid \bullet \psi \mid \psi \quad | \quad \psi U \psi \mid \neg \psi \mid \psi \lor \psi\)

The atomic propositions are types \(\tau\) of dataspace assertions. The basic syntax comes with two temporal operators: \(\bullet \psi\) (next) and \(\psi_1 \ U \psi_2\) (strong-until). The set of formulas includes negation (\(\neg \psi\)) and disjunction (\(\psi \lor \psi\)). Note Standard constructs such as conjunction (\(\psi \land \psi\)), eventually (\(\circ \psi\)), 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 90 (AP(DS)).

\[ \tau \in \text{AP}(\text{DS}) \text{ iff there exists } (c, \ell) \text{ s.t. } (c, \ell) \in R, \vdash c : \tau \]
where DS = \{[\ell, \Delta]; R; \overrightarrow{A}\}

\[ \tau \in \text{AP}(\text{DS}_T) \text{ iff there exists } \ell \text{ s.t. } (\tau, \ell) \in R_T \]
where DS_T = \{[\ell, \Delta_T]; R_T; \overrightarrow{A}\}

That is, an atomic proposition belongs to a dataspace configuration, if its table of assertions c contains an assertion (by any actor l) that type checks as \(\tau\). The notation is overloaded for type-level dataspace configurations, AP(DS_T). At the type level, an atomic proposition \(\tau\) belongs to the configuration if such a type is present in the configuration’s assertions (R_T).

The second step concerns the notion of satisfaction, that is, when a dataspace configuration—and its run (definition 8)—satisfies an LTL specification.

Definition 91 (DS \models \tau). For both term and type-level configurations,

\[ DS \models \tau \text{ iff } \tau \in \text{AP}(\text{DS}) \]
\[ DS \models \neg \psi \text{ iff } DS \not\models \psi \]
\[ DS \models \psi_1 \lor \psi_2 \text{ iff } DS \models \psi_1 \text{ or } DS \models \psi_2 \]
\[ DS \models \bullet \psi \text{ iff } DS_1 \models \psi \]
\[ DS \models \psi_1 \mathbf{U} \psi_2 \text{ iff there exist an } i \text{ s.t. } DS_i \models \psi_2 \text{ and } \forall j, 0 \leq j < i, DS_j \models \psi_1 \]

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 \(\psi\) holds starting from the next step in the run. Similarly, the other temporal operator, strong until (\(\psi_1 \mathbf{U} \psi_2\)), holds when \(\psi_1\) is true for some finite number of steps in a dataspace run, at which point \(\psi_2\) is true. All other syntactic forms have expected meaning.

It is now possible to define what it means for a program—a description of the initial facet-based actors in the dataspace—to satisfy a specification.

Definition 92 (\(\overrightarrow{Pr} \models \psi\)). \(\overrightarrow{Pr} \models \psi\) iff boot_{DS} (\(\overrightarrow{Pr}\)) = DS \land DS_1 \models \psi

This notion of satisfaction lifts to effect types describing facet-based actors.

Definition 93 (\(\overrightarrow{T} \models \psi\)). \(\overrightarrow{T} \models \psi\) iff boot_{DS_T} (\(\overrightarrow{T}\)) = DS_T \land DS_T_1 \models \psi

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.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Term Meaning</th>
<th>Type Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>light(Bool)</td>
<td>Either light(true) or light(false) is currently asserted in the dataspace.</td>
<td>light(Bool) asserted.</td>
</tr>
<tr>
<td>¬light(Bool)</td>
<td>Neither light(true) nor light(false) is currently asserted in the dataspace.</td>
<td>Light(Bool) not asserted.</td>
</tr>
<tr>
<td>⟨light(Bool)⟩</td>
<td>The execution reaches a light(Bool) configuration in a finite number of steps.</td>
<td>Same as term meaning.</td>
</tr>
<tr>
<td>□light(Bool) ⇒</td>
<td>Whenever light(Bool) is true, there is eventually no switch-on() assertion in</td>
<td>Same as term meaning.</td>
</tr>
<tr>
<td>(light(Bool)</td>
<td>the dataspace, and light(Bool) stays true until then.</td>
<td></td>
</tr>
<tr>
<td>U ¬switch-on()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7: Formal LTL Examples

Figure 5.7 illustrates how to translate the example specification from section 5.1.2 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 66). 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. The set safe(M) collects the first kind for each configuration M that does not error due to a partial primitive. The judgment safe(Pr) describes a facet description that bool_pr-s to a safe machine state.

The next element is the 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; they react to the same (types of) events; and their reactions to those events yield related states.

**Definition 94.** \( M \approx M_T \) if and only if:

- \( \vdash_\pi \text{assertions-of}_M(M) : \text{assertions-of}_{M_T}(M_T) \)
- \( \forall l \text{ such that } M \xrightarrow{l} M', \exists l_{r,T} \vdash_\pi l : l_{r,T} \text{ such that } M_T \xrightarrow{l_{r,T}} M'_T \text{ and } M' \approx M'_T \)
- \( \forall l_{r,T} \text{ such that } M_T \xrightarrow{l_{r,T}} M'_T', \exists l \vdash_\pi l : l_{r,T} \text{ such that } M \xrightarrow{l} M' \text{ and } M' \approx M'_T \)
Assuming the machine belongs to the set of safe ones, typing implies correspondence.

**Theorem 95.** If $\vdash_M M : M_T$ and $\text{safe}(M)$ then $M \approx M_T$.

**Proof** The initial machine states for a term and its type are related (lemma 97). The key meta functions preserve typing (lemma 99), and related events always exist (lemma 98).

Theorem 95 establishes a similarity between term and type level machines based on their inputs (injected events) and output labels (spawned actors). Lemma 103 shows that the similarity extends to the output assertions of $M$ and output (type) assertions of $M_T$.

The point of this theorem is that an actor machine and its type implement the same interface, up to typing. Thus, 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.

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 97);
- transitions in response to an external stimulus (lemmas 98, 99, 100); and
- transitions for performing internal work (lemmas 101, 102).

Finally, lemma 103 establishes that related type and term machines make related assertions.

Some of the lemmas make use of a function for directly assigning a type to a store.

**Definition 96 (Store Typing).**

\[
\begin{align*}
\text{type-store} : & \quad \sigma \rightarrow \sigma_T \\
\text{type-store}(\emptyset) & = \emptyset \\
\text{type-store}(\sigma[x \mapsto v]) & = \text{type-store}(\sigma)[x \mapsto \tau] \\
& \text{where } \vdash_v v : \tau
\end{align*}
\]

**Lemma 97 (Boot).** If

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

then

- $\text{boot}_T(T) = \langle \text{FT}_T; \ \overrightarrow{T}; \ \text{PS}_T; \ \pi_T; \ \emptyset \rangle$
- $\vdash_M \langle \text{FT}; \ \overrightarrow{T}; \ \text{PS}; \ \pi; \ \sigma \rangle : \langle \text{FT}_T; \ \text{ST}; \ \text{PS}_T; \ \pi_T; \ \text{type-store}(\sigma) \rangle$
Proof By induction on the type derivation and the soundness of the \textit{dispatch} meta-
function (lemma 99).

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

\textbf{Lemma 98 (Matching Events).} If

\begin{itemize}
\item \( \vdash P_v : \tau_P \)
\item \( \vdash c : \tau \)
\end{itemize}

\textit{then match}_P(P_v, c) = \gamma \iff \textit{match}_\gamma(\tau_P, \tau) = \gamma' \tau with \vdash \gamma : \gamma' \tau. \]

\textit{Proof} By induction on the pattern type derivation. \hfill \square

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

\textbf{Lemma 99 (Dispatch).} If

\begin{itemize}
\item \( \text{fid} = \langle \text{fn} \rangle \)
\item \( \Gamma \vdash_D D : D_T \)
\item \( \Gamma, \text{fn} : \text{FacetName}, \text{bindings}_D(D) \vdash_{Pr} Pr : T \)
\item \( \vdash_{Evt} Evt : Evt_T \)
\item \( \vdash_{\pi} \pi : \pi_T \)
\item \( \vdash_{\pi} \pi' : \pi'_T \)
\end{itemize}

\textit{then} \[ \text{dispatch}_1(\text{fid}, D, Pr, Evt, \pi, \pi', \sigma) = \overrightarrow{PS} \]

\textit{iff} \[ \text{dispatch}_1_T(D, T, Pr, Evt_T, \pi_T, \pi'_T) = \overrightarrow{PS_T} \]

\textit{with} \( \Gamma, \text{fn} : \text{FacetName} \vdash_{PS} \text{PS} : \text{PS}_T \)

\textit{Proof} By induction on the type derivations. The previous lemma (lemma 98) lifts to
events in general, so the term and type applications of \textit{match}_D yield related substitutions.

When applied to related event-handler bodies, the related substitutions yield related scripts.

In short, dispatching related events yields related scripts to execute. \hfill \square

\textbf{Lemma 100 (Partial Evaluation).} If \( \Gamma \vdash_{Pr} Pr : T \) and \textit{safe}(Pr) then \( p\text{-e}(Pr, \sigma, \text{fid}) = \overrightarrow{I} \sigma' \iff p\text{-e}_T(T, \sigma_T, \text{fid}) = \overrightarrow{I_T} \sigma'_T \) with \( \Gamma \vdash_I \text{I} : I_T \)
Proof By induction on the type derivation.

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

**Lemma 101** (Preservation). If $\vdash_M M : M_T$ and $\text{safe}(M)$ then $M \rightarrow M' = \langle FT; \overrightarrow{T}; PS; \pi; \sigma \rangle$ iff $M_T \rightarrow \langle FT_T; \overrightarrow{T}; PS_T; \pi_T; \sigma_T \rangle$ with $\vdash_M M' : \langle FT_T; \overrightarrow{T}; PS_T; \pi_T; \text{type-store}(\sigma) \rangle$.

Proof Following a standard approach [108], at most one of the machine transition rules can apply. Through the application of related lemmas, such as lemmas 99 and 100, 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 102** (Progress). If $\vdash_M M : M_T$ either inert$(M)$, inert$(M_T)$ or there exists $M', M'_T$ such that $M \xrightarrow{\|} M'$ and $M_T \xrightarrow{\|} M'_T$.

Proof By inspection of the machine state.

**Lemma 103** (Machine Assertions). If $\vdash_M M : M_T$ then $\vdash_{\pi} \text{assertions-of}_M(M) : \text{assertions-of}_{M_T}(M_T)$

Proof By induction on the type derivation and via similar properties for the $\text{assertions-of}$ family of functions.

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

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

**Theorem 104** (LTL Transference). If

- $\vdash_{Pr} Pr : \overrightarrow{T}$
- $\overrightarrow{T} \models \psi$

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

Proof Due to the simulation property (theorem 95), 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 [35, 54].

In detail, the 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, types cannot capture this notion.

If the system were to account for the possibility of unexpected exceptions, every program includes the degenerate behavior of just crashing. Even the addition of supervision actors would not avoid this collapse to a single-point model.

I conjecture that by generalizing work on exception analysis via type systems [110] it might be possible to extend the type system to reason about such cases precisely, at least for some limited class of programs.

### 5.4 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 the implementation.

#### 5.4.1 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 chapter 4;
- 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, without any modifications to the latter, just as the model predicts.

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 [24]. Implementing the surface syntax greatly simplifies this addition of actor syntax and yields an implementation capable of growing to production-readiness. It also facilitates the addition of a type checker.

The run-time system is the most sophisticated of the three layers. Figure 5.8 describes the key elements, which strongly mirror the data structures of the machine and the meta-functions of the formal model of section 4.2. 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, enqueueing 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.

(install-facet! name behavior) Creates a facet with the given name and behavior
(terminate-facet! fid) Terminates the designated facet
(install-endpoint! assertion-fn handler-fn) Adds an endpoint to the current facet
(schedule-script! thunk) Enqueues a script for the current facet
(dataflow-record-observation! field-id ep-id) Records a dataflow dependency from an endpoint to a field
(dataflow-record-damage! field-id) Marks a field as damaged via an update
(dataflow-repair-damage! dataflow-graph) Updates the endpoints that depend upon damaged fields

Figure 5.8: Run-time Support for Facets

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.

Rationale The dataflow network is implemented imperatively, even though sophisticated, purely functional implementations exists [21]. While this pre-existing implementation is well-suited for pure languages, facet-oriented programs already make use of mutation. Hence, the decision to use a simple imperative dataflow implementation, with moderate efficiency but an easily-understood evaluation order and cost model.

5.4.2 Implementing the Type Checker

The type checker is again implemented with the Turnstile meta-DSL [18]. 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"
```

Line by line, this definition reads the following way:

- The macro applies to syntax that matches the (start-facet id expr ...) pattern. The pattern expects an identifier and a non-empty 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 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 forms: 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 call to the install-facet! procedure from the run-time system consumes the name and a thunk that runs the results of elaborating the endpoint forms—in the specified order. These forms elaborate to calls to install-endpoint!.

2 For expressions that use the keyword start-facet and don’t match the pattern syntax/parse raises an error with an informative error message.
3 Any type errors discovered while checking ep will be reported as soon as they are discovered, aborting the rest of the elaboration of start-facet.
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 5.4.

Additional Features

The implementation includes several additional features with the aim of supporting a programming style similar to untyped faceted actors and minimizing the burden of type annotations, allowing interaction between typed and untyped actors, and easing the transition from untyped to typed dataspace programming.

**Type Annotations.** The implementation seeks to avoid the need for type annotations wherever possible. For example, a local inference system instantiates the type arguments for polymorphic function application.

The presentation in chapter 3 and section 5.2 suggests that each actor creation requires a type annotation describing its communication type. Writing down such communication types quickly becomes burdensome as the number of types of assertions in the program grows. Instead, the implementation supports synthesizing a potential communication type for each actor and dataspace program. When the programmer elides a communication type annotation for a `spawn` action, the type checker constructs a type based on the actor’s endpoints. When the program launches a dataspace without specifying the communication type, the system checks the communication types of each of the contained actors for consistency. The communication type of the dataspace becomes, essentially, the union of the types of the contained actors.

Due to these efforts, the implementation consistently requires type annotations in only a few places:

- Every function definition must annotate the expected type of its arguments.
- Field declarations, and derived forms (see figure 4.6), whose type is not the same as the type of their initialization expression.
- The `require/typed` form for importing values from untyped code with specific types.
- Providing a default type for each field when defining a new `struct` type.

Strictly speaking, `struct` fields do not require default types. However, they prove to be useful in avoiding type annotations when writing patterns over assertions for event handlers. When pattern matching on expressions, such as with `match`, the type of the matched expression provides a useful guide:
However, event handlers in actors perform *implicit* pattern matching. That is, there is no expression in the syntax representing the value being matched. The ubiquitous use of such patterns in facet programs demands the additional support of types for struct fields.

The example actor of figure 5.9 demonstrates two mechanisms for synthesizing the types of binding variables in event handler patterns (line 8).

```scheme
(struct milk-bid (b))
(define-type-alias MilkDataspace
  (U (MilkBidT Int)
      (Observe (MilkBidT ⋆/t))))
(define (seller [price : Int])
  (spawn #:type MilkDataspace
    (on (asserted (milk-bid $b))
      (when (>= b price)
        ...))))
```

Figure 5.9: Light actor with Boolean assertion

On the left, the definition of the milk-bid struct does not provide a default type for the price field. However, the developer may describe the communication type for the dataspace or particular actors within the dataspace (lines 2-4). By supplying that type on line 7, the same mechanism for instantiating types in explicit pattern matches, like the match example above, identifies that any successful matches of the pattern on line 8 associate \( p \) with type Int.

As mentioned above, the implementation mostly obviates the need for manually writing down communication types. The right side of the figure shows an alternative approach: adding a default type to the price field (line 1). Then, all pattern matches on milk-bid structs utilize the default field types when the context does not provide type information. Default field types also prove useful for writing down types. With the default provided, the struct definition automatically creates a type alias for MilkBid as (MilkBidT Int).\(^4\)

Finally, when greater flexibility is needed, binding variables in patterns may be explicitly annotated with minimal syntactic ceremony, as in

```scheme
(on (asserted (milk-bid $b:Int)) ...)
```

Larger types may also be referred with the \$ binding pattern constructor:

```scheme
(on (asserted (milk-bid ($ b (U Bool Int)))) ...)
```

\(^4\) When only some fields have default types, the "T"-less type alias is a type constructor that takes types for the default-less fields and uses the default types for the others.
Typed/Untyped Interoperation. Interoperation between actors written in the typed and untyped variants of dataspaces is important. It allows for typed programs to reuse existing functionality from untyped programs, such as the large number of driver actors for various forms of I/O. It also allows for a gradual, module-by-module transition from an untyped program to a typed one. In order to facilitate such interaction, typed and untyped programs must be able to share data structures. Towards this end, the typed implementation reuses all of the data structures from the untyped implementation, such as the structs for observe and message. Additionally, the require/typed form allows importing structs from untyped code and integrating them with type checking.\footnote{The syntax of the form is inspired by Typed Racket’s form of the same name \cite{typed-racket}.}

For example, the typed implementation of the web chat program (described in section \ref{sec:web-chat}) imports a number of data structures from untyped code, such as the account struct:

```
(requires/typed "../untyped/protocol.rkt"
[#:struct account [email : String]])
```

The #:struct clause of require/typed imports the struct definition for account from the specified module. The import includes the constructor, accessor functions (account-email), generates the corresponding type constructor and type checking rules, and records the default type of the email field, eliminating the need for many potential type annotations. With the import, typed actors may freely send and receive account structures to and from untyped actors.

In some cases, a field of a struct may not have a sensible default type, or the documentation may be lacking. In that case, the default type for the field may be omitted. For example, the web chat program uses the api struct to represent a variety of different kinds of requests. The value field of the import struct may then be elided, with the types from each use site giving rise to different types of requests:

```
(requires/typed "../untyped/protocol.rkt"
[#:struct api [session : Session] value])
```

The data definition for some untyped data may be quite complex, involving an arbitrarily large tree or graph of different kinds of structs and other forms of data. In order to prevent the typed developer from having to transitively describe the types of all dependencies, require/typed may create opaque types for portions of the data whose details are unnecessary to the typed code. The type checker gains no knowledge of how values of opaque types should be created or used; it simply knows they exist. For example, the web chat program utilizes a library providing a markup data type for working with various forms of hypertext. The program creates instances of markup in only a few locations, and, once created, never analyzes them. In this case, the typed program can import the type opaque, together with a constructor or two, and then freely pass around such values:
Finally, describing the communication type of an untyped actor (the assertions and messages it communicates with) allows importing functions that spawn such actors to typed code. The following listing shows the import of an untyped driver actor for performing I/O via stdin. The type assigned to activate! describes a function of no arguments or return values, but with the effect of spawning an actor with communication type StdInDriver:

```
(define-type-alias StdInDriver
  (U (Message LineIn)
     (Message LineOut)
     (Observe (LineInT ⋆/t))
     (Observe (LineOutT ⋆/t))))
```

```
(require/typed (submod "../std-in.rkt" syndicate-main)
  [activate! : (proc → ⊥ #:spawns ((Actor StdInDriver)))]
```

**Transitioning from Untyped to Typed Actors.** The process of creating a typed version of a program from an untyped program follows a straightforward recipe:

- Translate the data definitions to typed structs.

- Where the program utilizes external or complicated data, import it as an opaque type, together with any needed constructors and helper functions.

- Translate each helper function and actor definition one by one.

- For a small number of functions/subexpressions that are either untypable or would be exceedingly awkward to rewrite, extract it as an untyped helper function and import it using require/typed.

- A common pattern for actors is to define recursive, or mutually recursive, functions that behave like a state machine. Because the typed implementation does not permit (mutual) recursion, transform the explicit loop to an implicit one by generating and consuming events.

Once a functioning typed version of the program is reached, some additional transformations to facilitate behavioral checking may be necessary. The behavioral analysis allows arbitrary branching within the body of event handler endpoints, but some untyped programs make use of the dynamic nature of the implementation to branch while installing endpoints during facet start up. In such instances, the branches must be shifted from the body of the facet to the endpoint or the context, depending on the desired behavior.
5.4.3 Implementing the Model Checker Translation

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

\[
(\text{verify-actors spec actor-type ...})
\]

In terms of theorem 104, this formula requests a check of \( \mathcal{T} \models \psi \) where the LTL formula \( \psi \) corresponds to \( \text{spec} \), and \( \mathcal{T} \) is the series of \( \text{actor-type} \)s, one per actor in the dataspace program. If the check succeeds, then the actors of these types jointly realize the specification.

The implementation of \( \text{verify-actors} \) utilizes the model checker SPIN [56] to check such uses. That is, it translates each use of \( \text{verify-actors} \) into a Promela program, SPIN’s input form.

More precisely, the translation turns each \( \text{actor-type} \) 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 \( \psi \). If verification fails, the trace provided by SPIN is back-translated into the actions of the type-level actors.

Figure 5.10 displays a sample Promela program. It is the result of translating the Light actor’s type. Here is a line-by-line description:

- Line 1 declares an enumeration type for the possible combinations of active facets.
- Line 2 creates a global variable that controls scheduling.
- Line 3 names the Promela process that is active when the program starts.
- Line 4 uses a local variable to track the actor’s currently active facets.
- Lines 5-8 introduces local variables for tracking the assertions known to the actor.
- Lines 9-11 issue the actor’s initial assertions. The \texttt{ASSERT} macro updates a global variable representing the number of assertions of the given type. The block executes atomically to avoid any unnecessary interleaving with other actor’s initialization in the SPIN search space. Line 10 is a comment (/* ...*/) that embeds information from the Racket source program; this information is used to back-translate counterexamples of the SPIN model to the source level.
mtype = {dur_im_lgt_dur, dur_lgt, dur_lgt_dur, lgt, dur_im_lgt}
bool light_proc_clock = true;
active proctype light_proc() {
  mtype current = lgt;
  bool know_WallSwitchOnT_Symbol = false;
  bool know_RoomEmptyT_Symbol = false;
  bool know_RoomOccupiedT_Symbol = false;
  bool know_RoomAssignmentT_Symbol_Symbol = false;
  atomic {
    ASSERT(Obs_WallSwitchOnT_Symbol); /*#s(assert #s(Observe ...))*/
  }
  do
  :: true ->
  light_proc_clock == GLOBAL_CLOCK;
  atomic {
    light_proc_clock = !light_proc_clock;
    do
    :: current == dur_lgt_dur ->
    if
    :: RETRACTED(RoomAssignmentT_Symbol_Symbol) && /*#s(Retracted ...)*/
      know_RoomAssignmentT_Symbol_Symbol ->
      current = dur_lgt;
    know_RoomAssignmentT_Symbol_Symbol = false;
    RETRACT(Obs_LightOnCmdT_Symbol); /*#s(retract #s(Observe ...))*/
    RETRACT(Obs_RoomEmptyT_Symbol); /*#s(retract #s(Observe ...))*/
    RETRACT(Obs_RoomOccupiedT_Symbol); /*#s(retract ...)*/
    RETRACT(Obs_LightOffCmdT_Symbol); /*#s(retract ...)*/
    know_RoomOccupiedT_Symbol = false;
    know_RoomEmptyT_Symbol = false;
    :: ASSERTED(RoomOccupiedT_Symbol) && /*#s(Asserted ...)*/
      !know_RoomOccupiedT_Symbol ->
      know_RoomOccupiedT_Symbol ->
      ...
  }
}

Figure 5.10: Excerpt of Promela program for the SPIN model checker
• Lines 12-18 launch an infinite loop, corresponding the actor’s ongoing behavior: wait until the actor is enabled (line 14); start an atomic block (line 15); and dispatch based on which facets are currently active (lines 17-18).

• Line 19 dispatches to the event handlers within the active facets.

• Lines 20-21 represent the condition for one of the active event handlers, the retraction of a room assignment. The RETRACTED macro checks that there are no assertions of the corresponding type. In order for the event to fire, there must not be any such assertions and the actor must have seen such a prior assertion (line 21). Again, the comment embeds information relevant for the back-translation.

• Lines 22-30 perform effects dictated by the type of the event handler’s body. In this case, it stops a facet (updates current, line 22), updates the actor’s knowledge (lines 23, 28, 29), and retracts active assertions (lines 24-27).

• Line 31 is the beginning of the next event handler, which is elided.

The rest of the Promela process follows the same pattern.

Refrerring to Types in the Program. The expected use case is that a programmer will use the analysis to check the types of the actor code, as opposed to checking the behavior of hand-written types. For example, a typical program fragment might look like this:

```
(define (spawn-actor)
  (define my-id (gensym))
  (spawn
    (start-facet root
      ... implementation ...
    ))
  my-id)
```

The type of the actor’s behavior is associated with a particular sub-expression of the function definition. That behavioral type may be quite large, depending on the body of the actor, and not feasibly written by hand.

In response, the implementation supports two methods of referring to such behavioral types in verify-actors:

1. By referring to the name of the facet in question, in this case root. More technically, every facet name is associated with the corresponding effect type in the test submodule [33]. Consequently, facets that are the subject of behavioral analysis must have distinct names.

2. By writing an expression that has the desired effect. In this example, that would be written (spawn-actor). This could be potentially misleading, where the function argument expressions and their types are not actually used.
When actor types fail to satisfy the LTL specification, the developer must be given some explanation in order to debug the program or potentially revise the specification. A counter-example illustrating the trace of a violating program execution serves this purpose. Indeed, such a trace is exactly what SPIN produces when a check fails. However, the counter-example from SPIN is a trace of the generated Promela processes.

Figure 5.11 illustrates an excerpt of the information reported by SPIN. The information is unsuitable for reporting to the developer who wrote a program in Racket. The generated Promela program contains additional code for simulating dataspace programs, such as the clock variables that implement synchronization, and transformed versions of source-level concepts, such as types.

In order to report usable information, the implementation embeds source-level information in the Promela program as comments. The steps that correspond to relevant source-level steps are the Promela line numbers that have such a comment. By consulting the trace from SPIN and the information in comments, the implementation can reconstruct a trace in terms of dataspace actors and their actions. Figure 5.12 shows a portion of such a reconstructed trace.

**Interacting with the outside world.** SPIN analyzes the reachable state-space of the input program, so all behavior of interest must be exercised by the input. In other words, it views programs as closed systems as opposed to open programs that may interact with their environment. This poses a challenge for those dataspace
programs that do interact with the outside world via driver actors or a user interface. For example, consider the following actor:

```
(define-constructor* (go))
(define-constructor* (ack))
(spawn
  (start-facet A
    (during (go)
      (assert (ack))))))
```

A simple specification of its behavior would state the following implication:

```
(define-ltl spec
  (Always (Implies Go
           (Eventually Ack))))
```

With the following check succeeding:

```
(verify-actors spec A)
```

However, the check is vacuously true. For example, the following specification would pass as well:

```
(define-ltl bad-spec
  (Always (Implies Go
           (Always (Not Ack))))
```

The issue is that no state in the program satisfies the precedent of the implication, a go assertion, so by basic Boolean logic the implication is true regardless of the antecedent.

To remedy the situation, the implementation supports specifying types of assertions and messages that may appear in the program via I/O, using a keyword-argument to `verify-actors`:

```
(verify-actors spec
#:IO Go
A)
```

The invocation gives rise to an I/O process in the SPIN model that non-deterministically makes and retracts go assertions. The implementation supports a union describing all the possible types of assertions and messages that correspond to communication with the outside world. With the #:IO specification, checking spec succeeds while bad-spec fails, as desired.

The implementation chooses a particular model of the program’s environment, namely one that performs one action each time the program becomes idle. This model of the environment is ideal for some scenarios, but a poor fit for others. For example, a server intended to cope with high workloads ought to consider the case where many inputs arrive before a previous request finishes processing. Ideally, the programmer would be able to select from one of many I/O models, or even specify their own.
The implementation provides a form of bounded verification over a specific collection of actor behaviors. That is, the scope of analysis is limited to the behaviors provided as inputs to `verify-actors` and is poorly suited for analyzing certain dynamic collections of actors, such as arbitrarily-sized ring topologies. While the I/O model greatly increases the scope of the analysis, the number of potential behaviors is still bounded. Moreover, the programmer must take care to choose the right I/O types for each invocation to `verify-actors`. If the specified types do not exercise the desired behavior in the program, the analysis is susceptible to yielding vacuous results. Section 6.1 discusses the challenge of such vacuous results in further detail.

5.4.4 Type-Varying Assertions

One of the first limitations encountered in the type precision of the implementation was due to assertions containing union data. An experimental extension seeks to improve this shortcoming by adding expressiveness to the type language and greater reasoning power to the checker. This section describes the design and implementation of that extension, *type-varying assertions*.

The Problem. Many protocols for dataspace communication feature assertions carrying union data. For example, in the smarthome example, a light may communicate that it is currently on by asserting `(light-state #true)` or off by sharing `(light-state #false)`. The type of these assertions is `(LightStateT Bool)`. Figure 5.13 demonstrates a simplistic implementation of such an actor that updates its assertion in response to a `turn-on` command.

```
1 (define (spawn-light [initial-state : Bool])
2   (spawn
3     (field [curr-state initial-state])
4     (assert (light-state (! curr-state)))
5     (on (asserted (turn-on))
6       (:= curr-state #true))))
```

Figure 5.13: Light actor with Boolean assertion

One simple statement about the program’s behavior is that, after a `turn-on` command, the light should be on. Unfortunately, the type language as presented does not have the precision to describe this property, because the assertion of the light being off and being on have the exact same type, `Bool`.

The Solution. The first step towards a solution, especially considering that the type language already includes union types, is to define singleton types for `#true` and
false, True and False, with Bool becoming an alias for the union of the two. With this change, the following LTL formula describes the program’s behavior:

(Always (Implies TurnOn (Eventually (LightT True))))

With sufficient precision in the type language for stating the property, the challenge then becomes capturing enough information in the behavioral type of the actor for the checking process to correctly accept this LTL formula as a true property. In other words, the checker needs to be able to recreate the following reasoning:

1. The type of the light state assertion is (LightStateT Bool) at first, because the initial-state is an unknown Bool (line 1).

2. In response to a turn-on assertion, the actor updates the curr-state field, and the type of the actor’s current assertion changes to (LightStateT True) (line 6).

In other words, the field update on line 6 causes a change in the type of the assertion, also known as a strong update.

The implementation of this reasoning is a form of typestate [98]. The type of a field’s value may have a set of possible types. Field update operations may trigger a transition from one current type for the field’s value to another. When an assertion of an actor depends on the value of a field, as the one on line 4 does, the type of that assertion is a function from the current typestate of the field.

Discussion. In behavioral type systems, choices between different forms of behavior must be made explicitly in the form of a tagged message. That is, they force branches to depend on the constructor, or tag, of a message. However, in actual programs, important information that may call for branching is often carried by data within a message (or assertion). I have sought to accommodate this programming practice by allowing arbitrary, effectful branching (if) within expressions. Incorporating a form of typestate in this manner helps reconcile behavioral type system techniques with the way developers actually write programs.

5.5 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

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

Splitting numeric types into the union of Zero and NonZero also proves useful.
This section analyzes 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.

5.5.1 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. Each program consists of about 500 lines of code, comprising the concurrent core of a larger system.

smarthome. This program simulates a smarthome control system with several types of devices as well as a text-based user interface. Via this interface, the user can move between rooms, install and configure lights and presence sensors, and issue commands and queries.

- **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.\(^7\) 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.\(^8\) Clients submit jobs consisting of some number of inter-dependent 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

\(^7\) This property arose from 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.

\(^8\) [https://flink.apache.org](https://flink.apache.org)
manager, and then from a task manager to a task runner, and then 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 (DAG) 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.

- **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 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 [58]. 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.
Property | LTL
--- | ---
Task Delegation | (Always (Implies (Observe TaskPerformance) (Eventually TaskPerformance)))
Job Completion | (Always (Implies (Observe JobCompletion) (Eventually JobCompletion)))
Load Balancing | (Always (And (Implies (Observe MapTaskPerformance) (Not (Observe ReduceTaskPerformance))) (Implies (Observe ReduceTaskPerformance) (Not (Observe MapTaskPerformance)))))

Table 5.2: Data Processing LTL Specifications

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.** This is the property that the election completes.

- **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.

Property | LTL
--- | ---
Resolution | (Eventually Winner)
Cand. Misbehavior | (And (Eventually Candidate) (Eventually (Until Candidate (Or (M Tally) Winner))))
Voter Misbehavior (weak) | (And (Always (Implies Round (Eventually Vote))) (Eventually (Always Voter))))

Table 5.3: Caucus LTL Specifications
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:

  \[
  ;; \quad (\text{Observe LayoutSolution})^+ \implies \\
  ;; \quad \text{RequestedLayoutSize} \implies \\
  ;; \quad \text{ComputedLayoutSize} \land \text{LayoutSolution}^+
  \]

  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.

<table>
<thead>
<tr>
<th>Property</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout</td>
<td>(Always (Implies (Observe LayoutSolution) (Implies (Eventually RequestedLayoutSize) (Eventually (And ComputedLayoutSize LayoutSolution)))))</td>
</tr>
<tr>
<td>Menu Duration</td>
<td>(Always (Implies MenuItem (WeakUntil MenuItem (M (Inbound MouseEvent))))))</td>
</tr>
</tbody>
</table>

Table 5.4: Windowing System LTL Specifications

WEB CHAT. This program implements the server for a chat service loosely based on Slack.\(^9\) It allows a user to connect, sign up for an account, and then create and join

\(^9\) https://slack.com
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.”

<table>
<thead>
<tr>
<th>Property</th>
<th>LTL</th>
</tr>
</thead>
</table>
| Conversation Release | (Always
  (Implies (And UnansweredInvitation
    (Until UnansweredInvitation
      (Not Invitation))
    (Eventually (Not Question))))) |
| Contact Release     | (Always
  (Implies (Until (And PermReq Question (Not AnswerS))
    (Not PermReq))
    (Eventually (Not Question)))) |

Table 5.5: Web Chat LTL Specifications

131
5.5.2 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 5.14 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 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 5.14 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 or message ever arises. The following LTL formula states this property with respect to the friend1 actor’s interest in Time assertions:

\[
\text{(define-ltl friend1-deadlock (Always (Implies (Observe (TimeT */t))) (Eventually (Or Time (M Time) (Not (Observe (TimeT */t)))))))}
\]

This LTL property can then be checked against the implementations of friend1 and friend2:

\[
\text{(verify-actors friend1-deadlock f1 f2)}
\]

The check fails, as should be expected for figure 5.14, and produces a trace illustrating 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:

\[
\text{(define-ltl friend2-deadlock (Always (Implies (Observe (PlaceT */p))) (Eventually (Or Place (M Place) (Not (Observe (PlaceT */p)))))))}
\]

Generalizing, a program may be checked for deadlocks by taking each assertion of interest \((\text{Observe } \tau)\) and checking the following LTL formula:

\[
\text{(Always (Implies (Observe } \tau) \text{ (Eventually (Or } \tau \text{ (M } \tau \text{) (Not (Observe } \tau)))))}
\]

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:

\[
\text{(Eventually (And (Always (Observe } \tau) \text{ (Not (Eventually (Or } \tau \text{ (M } \tau))))))}
\]

This section refers to the former property as strong (soft) deadlock freedom and the latter as weak (soft) deadlock freedom. The check-deadlock-free and check-deadlock-free* forms check the two properties, respectively.

10 The \text{M} constructor in the LTL formula stands for a (broadcast) message in the dataspace.
5.5.3 Classifications

At the coarsest granularity, attempting to check each property of a program either succeeds or fails. Subdividing failures into further categories helps to improve the understanding of the limitations of this approach. Hence, the following categories correspond to different directions along which the checking framework could be improved to accommodate checking the given property.

**Type Precision.** The atomic propositions in LTL specifications describe the types of assertions and messages in the dataspace. The type system’s implementation provides a basic type language, allowing specifications to describe data constructors (structs) and primitive data such as integers, Booleans, and strings. The choice of type language places a limit on the expressiveness of specifications.

There are many approaches to increasing the precision of a type language [79, 80], of varying complexity. Accordingly, one outcome for checking a property is that it fails, but it would succeed with such extensions to the precision of types. In other words, it would only take a modest implementation effort to accommodate the property. The implementation already features one such extension (see section 5.4.4). However, such extensions are not often the outcome of a systematic design process. Piling on additional extensions to the type language is not the ideal way to build a system. As more extensions are added, each extension has to consider each other extension, and their seemingly “minor” nature begins to suffer from complex interactions.

By contrast, there is a space of properties that would require heavyweight type checking machinery in order to describe. For example, a full dependent type system would allow properties that describe fine-grain relationships between assertions. A dependent type system could easily describe many relationships between numeric assertions in a dataspace. Integration with a different form of solver, such as SMT, could support checking richer properties, as well.11

Additionally, the approach of only describing the types of assertions leaves many other properties completely inexpressible. For example, focusing on the presence/absence of types assertions leaves no way of talking about the multiplicity of assertions, the identity of an actor, or concepts such as fairness. Doing so may require creating a program logic for dataspaces and facets and integrating it with the implementation.

In theory, it is perfectly possible to imagine a design that incorporated such capabilities (dependent types, SMT solvers, a program logic). The implementation of such a tool, or extending the current one in such a fashion, would be a major undertaking beyond the current effort. Thus, the categorization distinguishes properties that could be checked with a “minor” extension from those demanding a “major” departure.

---

11 Liquid types [83, 105] offer a blueprint for integrating such solvers with a more conventional type system.
TEMPORAL PRECISION. Using linear temporal logic as the language for specifying behavioral properties is another potentially limiting factor. When considering the state space of a program, for each state, LTL always considers all possible successor states together. For example, consider the state space of the program in figure 5.15. After starting, the program enters a loop, which it performs some unknown number of times, before exiting the loop to reach the finish state.

Now consider whether, starting at the state loop, the LTL proposition •finish, saying that the next state is finish, holds. The property does not hold, because the set of possible next states is \{loop, finish\}. That is, the next state is possibly not finish. Going further, taking start as the initial state, the LTL formula ◯finish does not hold, because it is always possible to continue looping.

One sensible question to ask about the program is whether it is possible to reach the finish state. Computation tree logic, or CTL for short, is a branching-time temporal logic that describes paths through a program’s state space. The CTL formula EF finish states that there exists a path that eventually reaches the finish state. The property holds of the program starting from both the loop and start states.12 Similarly, the CTL formula AG finish states that all paths from the current state reach a point where they always stay in the finish state.

The debate between branching and linear time in temporal logics is a recurring theme, with compelling arguments on both sides [104]. Many consider LTL the intuitive language for describing and reasoning about properties of programs, while some branching-time logics yield more efficient implementations for model checkers. The evaluation may indicate that a different temporal logic would be a better fit for dataspace programs. There are model checkers that support CTL and other branching time logics such as the modal μ-calculus. Changing the specification language for behavioral properties from LTL to, say, CTL would be tantamount to adding a new compiler backend.

PERFORMANCE. A final possibility is that, even if the implemented type language and temporal logic allows the expression of a desired property, the model checking process does not yield a result within a reasonable amount of time. The potential for a program’s state space to grow exponentially is unavoidable. Eventually, even optimized checkers such as SPIN 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.

12 In fact, every state is a suitable initial state for the property.
5.5.4 Results

Table 5.6 summarizes the results. Each row of the table describes an important 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. A ✗ in a different column indicates the primary limitation(s) preventing support for that property. In some cases, multiple approaches are possible, hence multiple columns have a check. A ✗ in the “Performance” column indicates a failure to terminate. A row with a ✓ in the “Checkable” column and an ✗ in the “Type Precision (Minor)” column indicates that checking the property made use of a non-standard extension to the implementation, such as the support for type-varying assertions (section 5.4.4). Section 5.5.5 discusses the reasoning behind each mark-up.

Performance

Table 5.7 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 SPIN, 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.
<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
<th>Expressible</th>
<th>Checkable</th>
<th>Type Precision (Minor)</th>
<th>Temporal Precision</th>
<th>Type Precision (Major)</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smarthome</td>
<td>Light Presence</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Steady State</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Temperature</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deadlock Freedom</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Task Delegation</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing</td>
<td>Job Completion</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Job Completion</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Weak)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Balancing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Balancing</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(weak)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deadlock Freedom</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caucus</td>
<td>Resolution</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resolution (Weak)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cand. Misbehavior</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Voter Misbehavior</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Voter Misbehavior</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(weak)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deadlock Freedom</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windowing</td>
<td>Layout</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>Menu Duration</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web Chat</td>
<td>Conv. Release</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conv. Release</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(weak)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cont. Release</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cont. Release</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(weak)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Evaluation Results
The biggest bottleneck in the system is macro expansion and type checking. As mentioned, the implementation uses the Turnstile DSL to realize the “type checking as macro expansion” technique. This approach allows for quickly creating a prototype of a typed language from typing rules, using the existing untyped implementation of dataspaces and facets in Racket. However, the performance ramifications turned out to be severe. Table 5.8 provides the time to compile each program, as well as the peak memory usage during compilation. Taking 20-45 seconds to compile a 500 line program is unacceptable. This aspect of the implementation is not fundamental, like the model checking. Clearly, one aspect of future work is implementing the type system in a conventional manner.
5.5.5 Interpretation and Discussion

This section discusses each of the properties that the implementation failed to support and conjectures what support for that property could look like. In a few cases, interesting details for successfully checked properties are provided as well.

SMARTHOME.

- **Steady State Temperature.** With a naive approach, the implementation is able to state this property, but not able to check it. The difficulty is that the notion of “reaching the target” depends on the outcome of integer arithmetic and comparisons. Integrating a numeric solver into the checker would allow reasoning about them. However, incorporating an assertion to the thermostat actor that announces when it has reached the target temperature allows the property to be checked with the existing implementation.

DATA PROCESSING.

- **Job Completion.** The implementation is not able to check this property. The loop in the protocol poses a significant problem to the checker. There are (at least) two potential ways to remedy the situation. By utilizing a branching time temporal logic, such as CTL, the specification could only require that that there always exists a path towards completing the job. Alternatively, the finite nature of processing the job DAG could be incorporated into the type system (say, via, an inductive elimination form).

- **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. Thus, the specification language would need to be enriched with a method for describing the acceptable number of 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 the property encounters the same difficulty as other protocols with loops. 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). Since LTL always talks about all possible executions, the specification language cannot even express that it is possible to reach the desired end state.

• **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 5.5.2 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 implementation of the candidate actors only listened for the intermediate results, not the winner announcement, thus 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.

In general, I think it is best to not worry too much about the particulars of scheduling actors within dataspaces, as long as the mechanism is fair (which it is). 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.
5.5.6 Threats to Validity

There are several threats to validity of this evaluation. The corpus comprises a small number of small programs. The programs in the corpus were authored by only three individuals, including myself. In selecting programs for the corpus, I considered factors that could affect the outcome. For example, I sought programs whose implementations involved a lot of concurrency and communication, as opposed to business logic or other concerns. When it comes to analyzing the properties of these programs, as the implementor of the behavioral checking tool, I am highly familiar with both its capabilities and its limitations. This knowledge likely biased me to search for properties that are a good fit for its capabilities.

5.6 Related Work

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 seek to 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 [30] 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 any use. Application-specific information could potentially be recovered by utilizing more powerful type-level reasoning, such as dependent session types [101], but presently bringing such machinery to bear in a usable programming language remains unsolved.

This work specifically draws inspiration from studying the techniques employed in the area more so than the end products. In that sense, it follows that of Igarashi and Kobayashi [60] and Chaki, Rajamani, and Rehof [17], 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 and Kobayashi, the checks are generic. Subtyping and other parts of the type checker may be instantiated to yield checks for data races, deadlocks, etc. Chaki, Rajamani, and Rehof 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 [89] 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, this approach allows checking any LTL formula, allowing application-specific correctness properties. The biggest difference to this work is the communication paradigm. While Effpi uses message-passing along channels in a process calculus, facet-based actors share knowledge via a dataspace.

Conversation Types. Conversation types [12] add behavioral checking to the Conversation Calculus [106]. A conversation type describes a sequence of message exchanges within a particular context, i.e., conversation. Like the global types of Honda, Yoshida, and Carbone [58], 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 Objects. 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 [51]. 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. The behavioral checks of facet programs instead treat effect types as abstracted, but still executable, processes, with the goal of (model)checking properties of programs.

The Rebeca modeling language [97] is the closest to work in the realm of active objects to this behavioral checking for facets. With Rebeca, a programmer may use communication and abstraction facilities of active objects to define a models of their program or system. The Rebeca model checker, Modere [61], can then verify properties of the model specified as temporal logic formulae. Rebeca has numerous extensions, notably broadcasting [111]. The goal of my work is to check properties of programs versus models of programs. It may be possible to target Rebeca as a backend for the implementation, rather than SPIN, and benefit from its message-passing-specific optimizations.

Types for the Join Calculus. The join calculus [36] 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 [37]. Recent work [23] has developed behavioral types for the join calculus around the notion of type state [98]. These behavioral types describe the kind 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, implementing such connectives as well as the necessary substructural support is challenging.

**Mailbox Types.** Similarly to the behavioral type systems for the Join Calculus, mailbox types [112] 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 [57] in particular have been utilized to describe and verify the communication properties of many kinds of systems. The methods used in this chapter are closely related to the generic π-calculus type system of Igarashi and Kobayashi [60]. Session types relate processes with abstract process types, including the primary process constructors: channel send and receive, parallel composition, and so on. The system is parametric over constraints on process types, allowing different instantiations targeted for different properties, such as deadlock prevention.

Multiparty session types [58, 88] 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.
CONCLUSION

This dissertation demonstrates the potential for type-based reasoning about concurrent dataspace programs, from the contents of individual utterances to temporal properties of conversations. Returning to the thesis that

*a domain-specific language for programming dataspaces actors enables type-based reasoning and automatic verification of program behavior,*

the foundational structural type system from chapter 3 together with the facet DSL described in chapter 4 and the behavioral checks of chapter 5 confirm the thesis. The evaluation from section 5.5 demonstrates that beyond being theoretically possible, this approach may have practical merit. The combination of structural types for describing communications, assertions and messages, and a temporal logic for stating specifications, suffices to describe interesting properties from several realistic programs and check them in a reasonable amount of time.

6.1 FUTURE WORK

This dissertation suggests several directions for future research on dataspace programming and reasoning:

**Improving the implementation.** The evaluation of section 5.5 points to several ways the implementation could be improved.

**Expressive power.** The first direction involves increasing the precision of behavioral specifications. Experience suggests that incorporating a limited form of dependent types, arithmetic reasoning, or induction principles could support a significant range of properties without drastically increasing the complexity of the implementation.

The evaluation also suggests that some desirable properties, such as the possibility of reaching a particular state in a protocol, are not expressible in LTL. Branching-time logics however, such as CTL, suit exactly such a purpose. Incorporating such a branching-time logic as a supplemental tool could be useful. The challenge is primarily one of implementation.

**Analyzing faults.** One particular challenge of concurrent programming is dealing with the possibility of *partial failure*, where a subset of components crash but the rest
continue operating. Indeed, one of the primary motivations for the design of the dataspace theory and facet notation is to equip developers with a powerful tool for dealing with such occurrences. However, as noted in section 5.3, allowing for the possibility that any actor may cause an exception is incompatible with the form of verification from chapter 5. That is, every program would include the behavior where every actor always crashes immediately after booting. In that scenario, the system does not satisfy any LTL specification.

The I/O modeling aspect can serve as an inspiration for an alternative approach to dealing with crashes. The approach to modeling I/O demonstrates that considering the full complexity of a problem is not always necessary for performing a useful analysis. In that spirit, a particular crash model may be considered for a program. The models that failures only occur within a certain subset of actors or every so often seem potentially useful. Indeed, the proper strategy may be to allow the programmer to select from a number of such models, or specify their own.

For example, suppose a developer wanted to check how failures in a financial transaction scenario play out. Specifically, assume that a client actor crashes and thus causes the dataspace to tear down its assertions, including an assertion about a pending transaction. In response, the system re-instantiates the client actor. The new client actor must determine what, if any, action is needed to complete the pending transaction. Any mistake in this recovery behavior may lead to rare but pernicious errors, such as duplicating or dropping the transaction. Using a failure-specific crash model enables a developer to analyze the actor’s behavior via model checking and may become an attractive approach to ensuring correctness in the face of such crashes.

**System Evolution.** As many real world systems grow, they often deal with the challenges that come with rolling out updates to an online system. This problem calls for ensuring both backward and forward compatibility between versions of code, including dealing with message and assertion formats (types) that change between versions. It is essential that any formal checking of the system, such as presented in this thesis, continues to work for the system during this phase of its lifetime. Indeed, as the complexity grows, it ought to become especially important to have reasoning aids.

**Usability, Alternative Logics.** The current implementation is essentially a compiler with a back end that supports a single target for model checking, SPIN. Like traditional compilers, the SPIN back-end consists of a few pieces: an execution model that encodes the behavior of facets and dataspaces in the behavior of Promela processes; a translation from the surface syntax and execution model to the target; and a runtime support system that completes programs executing at the target level. In total, the SPIN back end that implements this translation from an intermediate graph representation of facet-based actors is only a few thousand lines of code.
Utilizing a different model checker that supports a branching-time logic, such as mCRL2 [49], amounts to implementing another back-end translation. Notably, a significant portion of the implementation—the front end and the module that creates the graph representation (a few thousand lines of code each)—could be reused. The biggest challenge is the design of the execution model for facets and dataspace. The rest of the implementation process is a significant but not overwhelming undertaking.

Continuing the analogy to traditional compilers, there is no reason why the implementation could not simultaneously support multiple back ends. The front end could then support both specification languages, say LTL and CTL. In that spirit, the decision is shifted from the language implementor to the developer, who can decide which notion is the right fit for each property.

**Usability, Flaws in Specifications.** On the topic of specifications, the experience of using the behavioral checker and performing the evaluation of section 5.5 exposed a major shortcoming in the workflow: the first specification is never right. If the program violates the incorrect specification, the implementation provides a counterexample that can help diagnose the mistake. However, if the faulty specification happens to be true, then the tool reports success—and no other information. The situation is indistinguishable from the desired outcome of verification.

This problem concerns all of verification, especially considering the difficulty of both formulating and comprehending LTL propositions [48]. In many situations the best option is to perform ad hoc, manual sanity checks. Lamport discussed this problem and solution in the context of verification using TLA+ (emphasis added),

> When you write a specification, it usually contains errors. The purpose of running TLC is to find as many of those errors as possible. We hope an error in the specification will cause TLC to report an error.

> It’s a good idea to verify that TLC is performing the liveness checking you expect. Have it check a liveness property that the specification does not satisfy and make sure it reports an error.

> … make sure that TLC finds errors in properties that should be violated. [65, §14.5.2, §14.3.5, §14.5.3]

Since that time, researchers have made progress on automating such sanity checks [64] by analyzing a specification for vacuity [5] and satisfiability [85]. In another direction, Ammons et al. [4] use the idea of clustering executions to allow the developer to analyze the correctness of the specification and model. Unfortunately, these ideas are not yet widely adopted by model checkers. In particular, SPIN does not have any such support, leaving users stuck employing the methods described by Lamport.
In short, incorporating such assistance for developing specifications is a major priority for enhancing the usability of the implementation. Given that SPIN does not provide any such tools, doing so with the current setup is a significant challenge.¹

A TYPE FOR DATASPACE CONVERSATIONS. The behavioral types of chapter 5 describe the operations of a single actor. Taken together, the types describe the behavior of a collection of actors. The analysis and implementation of behavioral properties takes place after the program has been written. Thus, the current implementation misses out on one of the major utilities of a type system: guidance and assistance during the design phase of programming.

The situation calls for a language suitable for describing dataspace conversations from a global perspective. The end product might resemble the global type from multi-party sessions [58], choreography descriptions [14, 107], or the conversation calculus’s conversation types [12]. Two distinctive features of dataspace communication make adapting such designs a major challenge. First, at the programming level, entities are anonymous, raising the question of how to refer to the participants in conversations. Second, dataspace assertions have temporal duration, raising a notational challenge for incorporating descriptions of when they should appear and disappear.

EVALUATING DATASPACES AND FACETS. While section 5.5 presents an evaluation of the behavioral analysis of facet programs, the case for using facets and dataspaces as a concurrency theory remains largely rhetorical. In his dissertation, Garnock-Jones showed that facets obviate a number of programming patterns from other settings [40, §9]. The experience of programming with facets has illuminated several powerful characteristics:

- automatic demultiplexing of events;
- automatic synchronization of public and private state;
- localization of state, improving spatial locality and reducing the need for aggregation; plus
- easily composed behaviors, including the ability to create abstractions.

These traits suggest the design of an empirical evaluation for facets: implement the same program in several different theories (such as Akka, Elixir, and Go), and analyze the cost of those concerns in each program.

¹ Since satisfiability and vacuity reduce to the problem of model checking for LTL [84], one potential option is to make use of that reduction and utilize SPIN for just the checking.
6.2 CONCLUSION

Reflecting more broadly, this research underscores the power of DSLs to enrich programming. The same concurrency theory—the dataspace model—can lend itself to simple, structural reasoning (chapter 3) or automated behavioral verification (chapter 5), depending on the design of its linguistic interface. A key observation is that the facet notation reduces expressive power relative to a procedural interface, yet is both more attractive for developing programs and performing program analysis. In short, expressive power is good—but it has to be balanced with reasoning power.

Moreover, this research project is possible because of a suite of existing DSLs. The Turnstile library for Racket allowed for a tight prototype-and-experiment feedback loop for typed dataspace actors. Turnstile itself makes wide use of the syntax-parse DSL for writing syntax extensions. The SPIN model checker provides a DSL, Promela, for describing the behavior of concurrent systems. The abstraction provided by the Promela language allows projects such as this one to leverage the years of research and implementation effort spent creating an optimized model checker. In sum, my dissertation again makes a case for language-oriented programming [25].
BIBLIOGRAPHY


[38] Simon Fowler, Sam Lindley, and Philip Wadler. “Mixing metaphors: actors as channels and channels as actors.” In: (2017).


