Synthesizing Test Controllers from Types: Property-Guided Bug-Finding for Distributed System Models

ANONYMOUS AUTHOR(S)

Effective testing of distributed system designs is challenging. This is because the executions that lead to violations of important safety or liveness properties represent an infinitesimally small fragment of the set of all possible behaviors the system can exhibit. In this paper, we address this challenge by proposing a technique that automatically synthesizes a *test controller*— a program that guides the search for buggy executions— tailored to the model of a distributed system-under-test (SUT) and the property whose violation we are interested in triggering. We focus our solution on *open* systems in which the test controller must govern both the construction of messages injected into the SUT by an external environment as well as the order in which messages within the SUT are sent and received. Our approach rests on two technical innovations: first, we develop a novel trace-based refinement type system called *Prophecy Automata Types* that describes both the history of the system and its future behaviors using a symbolic variant of linear temporal logic. Second, we use these types to design a synthesis algorithm that constructs a program in a DSL tailored for expressing test controllers. Such programs directly express faulty executions in the target system by fixing the order in which messages are communicated among actors, and the contents of messages sent from an external environment to trigger component actions. We describe the implementation of our approach in a tool, Clouseau, and present a comprehensive evaluation on a set of diverse, non-trivial benchmarks, including a case study of an application model developed by a major cloud vendor, to justify our technique.

1 Introduction

Testing a model of a distributed system can help to find flaws early in the development cycle [2]. Frameworks like P [9, 10], for example, allow designers to write executable models whose behaviors can be explored using systematic testing methods (e.g., bounded model checking). Models are expressed in P as (reactive) asynchronously communicating state machines (or actors) that implement high-level logic, but which abstract away low-level details that a concrete implementation must address. Actors are responsible for sending and responding to messages from other actors, or from messages sent by an external environment, i.e., by clients or other actors not under test. Reasoning about a system's design thus typically involves (1) providing definitions for the actors that comprise the model; (2) defining an environment that closes the system by generating inputs to trigger behaviors; and, (3) providing a specification that the model should satisfy.

In this setting, the goal of a testing framework is to explore all possible executions of the closed system derived from the composition of (1) and (2) that can violate (3). In deciding how to perform this exploration, we must consider (a) how the actors of the system-under-test (SUT) interact with the environment and each other (e.g., the messages they generate in response to other messages), (b) what messages are generated by the environment, potentially in response to outputs produced by the SUT, and, (c) the order in which messages generated by actors are received and handled by others. For example, choosing to control delivery of a message sent from one actor to another can be used to simulate a weak consistency semantics [42] in a model of replicated state. Note that (a) captures how messages are handled whereas (b) captures the order in which messages are handled.

A test framework uses a *controller* to answer the last two questions. Specifically, the controller consists of both (1) an input generator that provides input messages to the actors under test, thus closing the open system and (2) a scheduler that controls the ordering of messages sent and received by actors in the closed system. Controllers typically implement either a random or enumerative exploration stategy. Although conceptually simple, these approaches make it problematic to *a priori*

2024. ACM XXXX-XXXX/2024/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

determine if the testing framework will be effective in finding a model-specific design bug, given the typically very large state space of feasible executions that may have to be considered. Rather than having the controller undertake exploration for inputs and message orderings without any foresight on the property that we seek to violate, this paper investigates an alternative approach that specializes the actions the controller performs, explicitly guided by this property.

There are two immediate challenges that need to be overcome to realize this goal. First, we need to provide specifications expressive enough to capture interesting kinds of input constraints and message orderings (i.e., those relevant to the behaviors the model is expected to exhibit). Second, we need some way to leverage these specifications to appropriately bias our search procedure towards executions that are likely to evince a violation of a desired behavior. In this paper, we present a unified solution to both these challenges. The result is a novel framework for testing distributed system models, driven by a *bespoke* controller expressed as a program written in a DSL designed for this purpose. The controller is *automatically synthesized* from specifications capable of defining scheduling and input constraints provided by the model designer. A controller thus encodes a set of executions that can violate the target property, depending on the specifications and actions of the actors in the SUT. A concrete execution is produced by iteratively choosing inputs for environment-generated messages and observing how the actors in the SUT respond.

To enable controller synthesis, we equip actors with rich specifications in the form of *prophecy automata types* (PATS), a new form of type abstraction that augments refinement types with automata that describe programs with opaque internal state [47]. Our PAT-based specifications serve dual purposes, describing both (a) how the current global context impacts how a message is handled, and (b) how executing a message informs future actions the system can take. PAT automata are acceptors over LTL_f, the language of linear temporal logic over finite traces; notably, this language is equipped with efficient decision procedures [6], enabling our synthesis procedure to be highly-automated. Intutively, while each actor implements its own (potentially complex) internal logic, testing behaviors of the entire system requires exploring how these individual programs interact; PATS capture temporal and data dependencies between the messages that define these interactions.

To ground the discussion, consider how an actor that maintains a simple key-value store might respond to a message $\mathbf{getReq}(k)$ asking for the value of a key k. Because of the inherent asynchrony in the way requests and responses are handled, we can expect that after receiving this **getReq** message, the actor will respond with a $\mathbf{getResp}(k, v)$ message at some arbitrary point in the future; this response message holds the value v associated with k. In any reasonable implementation, v should be the same as *some* value the actor stored in response to an earlier message. We can encode the dependencies between these three messages via the following PAT:

$$\underbrace{[\Diamond\langle \mathbf{putReq} \mid k = \mathsf{key} \land v = \mathsf{val}\rangle]}_{\texttt{history automaton}}\underbrace{[\mathcal{S}\langle \mathbf{getReq} \mid k = \mathsf{key}\rangle]}_{\texttt{current automaton}}\underbrace{[\Diamond\langle \mathbf{getRsp} \mid k = \mathsf{key} \land v = \mathsf{val}\rangle]}_{\texttt{prophecy automaton}}$$

This type is parameterized by two variables, `key` and `val`, and is comprised of three automata; two of these use the eventually operator $\Diamond$, standard in temporal logics, to express temporal dependencies between messages. The first automaton specifies the *history* of messages that occurred prior to the handling of a **putReq** event. This specification captures any trace that has stored the value `val` in key `key`. The second automaton describes the *current* event, captured in this case as a singleton trace consisting of a **getReq** event over the key `key` (captured via the singleton modality $\mathcal{S}$). The traces that may follow this event are described by a *prophecy* automaton that stipulates that a **getRsp** message whose input contains the key `key` and value `val` will eventually appear, thus guaranteeing that every **getReq** message is paired with a **getRsp** message that returns some written value.

Intuitively, this type only ensures eventually consistent (EC) guarantees [3, 42], since the store is free to buffer and respond to read and write requests arbitrarily. While performant, this policy can be too permissive for users, who may expect the store to be strongly consistent (SC), i.e., one that always returns the value of a key at the point a request message is handled. We can specify this safety property as the following LTL$_f$ formula:[1]

$$\neg(\langle \textbf{putReq} \mid k = \mathsf{key} \wedge v = \mathsf{val}\rangle \wedge \bigcirc(\neg\langle \textbf{putReq} \mid k = \mathsf{key}\rangle \;\mathcal{U}\; \langle \textbf{getRsp} \mid k = \mathsf{key} \wedge v \neq \mathsf{val}\rangle))$$

Observe that probing if a store is SC cannot be done by testing how the actor maintaing the store handles these messages in isolation: a violation of SC crucially depends on a specific sequence of get and put messages with appropriate inputs. The above specification identifies an erroneous execution of the SUT as one whose last **putReq** binds key to val but in which a **getRsp** message on key generated in response to a previously issued **getReq** message returns a value other than val.

Our tool, Clouseau, generates executions that can test the behavior of distributed system models by synthesizing a controller program consistent with the specifications provided for handlers, but which systematically drives executions to violate a global safety or liveness property. Different executions of the controller program enforce the same ordering of message delivery and receipt, but allow the contents of messages that are generated from the environment to vary. Message contents can potentially influence dataflow within the actors that receive them, and thus the outputs they produce. We leverage Pat specifications to implement a top-down, component-based synthesis algorithm [13, 15, 16] which constructs a bespoke controller program that models messages as invocation of events (e.g., **putReq**). Traditional top-down synthesizers decompose the problem by first selecting a candidate component (e.g., a library method) and recursively synthesizing its arguments, using a component's specification to constrain the space of candidate arguments. In our setting, however, determining the appropriate handler to use while synthesizing a controller depends on both the messages that precede it and the requirements of the handlers for the messages that follow it. Our synthesis algorithm thus uses the data-dependent temporal relations defined by Pats to guide the search for a controller program. This program denotes a set of concrete traces in the SUT that should be explored. Each execution determines a fixed order in which trigger messages are sent from the environment, and sent/received by the model's actors. As it executes, the controller instantiates concrete values for environment messages, to yield a concrete schedule.

This paper makes the following contributions:

(1) We formalize a new symbolic trace-based type-guided component synthesis algorithm for representing sets of feasible schedules and message inputs in open reactive distributed system models. The output of the algorithm is a program written in a DSL tailored for expressing test controllers that governs executions in terms of message actions among the actors under test and the interaction of these actors with an external environment.
(2) To guide this algorithm, we propose Pats, a new type abstraction that allows the specification of temporal actions in terms of histories and futures over symbolic traces.
(3) We formalize a type system based on Pats and use it to relate the set of executions admitted by the synthesized controller with the actors under SUT and the target property.
(4) We describe Clouseau, a tool that realizes these ideas, and present a detailed evaluation that uses a diverse set of non-trivial, realistic benchmarks, including a case study drawn from an application model developed at a major cloud vendor. To the best of our knowledge, Clouseau is the first synthesis procedure capable of generating controllers from application-specific handler and safety constraints to guide testing of real-world distributed models.

The remainder of this paper is organized as follows. The next section introduces a running example, and use it to illustrate the ingredients of our approach. Sec. 3 defines a core distributed modeling language in which controllers are written and describes its type system. Our synthesis algorithm is described in Sec. 4. We discuss our implementation and our benchmark results in Sec. 5. Related work and conclusions are given in Sec. 6 and Sec. 7, resp.

[1]This specification uses two additional standard temporal logic modalities: $\bigcirc\,\phi$ requires that ϕ holds at the next step in a trace, and $\phi_1 \,\mathcal{U}\, \phi_2$ requires that ϕ_1 holds at every following point in a trace until ϕ_2 becomes true.

2 Overview

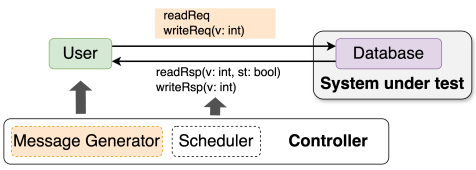

Fig. 1. A simplified database access workflow.

To motivate our approach, consider how we might test a (highly simplified) model of a distributed database application depicted in Fig. 1. This system includes two actors: a user and a database (the system under test). The user issues read and write messages to the database, while the implementation of the database persists user-supplied writes and responds to user requests to read its contents. To further simplify the example, we assume the database manages a single integer-valued record that users can read and write. As with the earlier key-value store example, messages are asynchronous and separated into two categories, one for requests and another for the corresponding responses. To handle a **writeReq** message, the database buffers the request, eventually persists its contents, and subsequently sends an acknowledgement of this fact via a **writeRsp** message to the user. The response to a **readReq** message is a **readRsp** message with two fields, v and st: when the st field is true, field v contains the value of the key at the time when the response was generated; a false status indicates that there is no value for the key in the database.

Our goal is to derive a controller that schedules messages to/from the database and determines the contents of messages generated by the user to the database. This process is independent of the precise logic defined by the actors in the SUT— rather, we rely on specifications of the relationships between the messages the actors send and receive. We group messages into two categories: in our example, messages sent by the user (e.g., **readReq** and **writeReq**) are independent of any prior messages and any actions taken by other actors, and can then thus be freely created and sent by a controller in some arbitrary order. We refer to such messages as *generable*. In contrast, messages sent from the database back to the user can only be produced in response to having received other messages. As the controller can only indirectly trigger such messages (and their contents), we refer to them as *observable*.

Traces and safety. An executable model generates a sequence of concrete messages, which we refer to as a *trace*. For our running example, we expect the database to satisfy a *read-your-writes* (RYW) policy [42] in which reads must see the most recent write successfully persisted. Under a database that provides EC semantics, however, users might witness the following trace:

$$\mathbf{writeReq}(3); \mathbf{writeReq}(4); \mathbf{writeRsp}(4); \mathbf{readReq}; \mathbf{writeRsp}(3); \mathbf{readRsp}(4, \mathtt{true}) \qquad (tr_1)$$

The trace reflects the order in which requests sent by the user are handled by the database, and responses generated by the database are received by the user.

In this trace, the **readRsp** message is received by the user from the database in response to a previously issued **readReq** message, but notably its contents contains a value other than the most recently persisted write. This can happen, for example, if messages on **writeReq** events are not guaranteed to be serviced in-order, or when the database state is replicated and the effect of the **writeRsp**(3) event has not been propagated to the replica that responds to the **readReq** message.

The following symbolic LTL$_f$ formula [6] formally captures a violation of RYW:

$$\Diamond(\langle \mathbf{writeRsp} \mid v = \mathtt{x}\rangle \land \bigcirc(\neg\langle \mathbf{writeRsp} \mid \top\rangle \,\mathcal{U}\, \langle \mathbf{readRsp} \mid v = \mathtt{y} \land st = \mathtt{true} \land \mathtt{y} \neq \mathtt{x}\rangle)) \qquad (A_{\mathtt{violateRYW}})$$

Here, $\langle \mathbf{writeRsp} \mid v = \mathtt{x}\rangle$ describes a set of messages, one for each possible concrete instantiation of x; we refer to this set as a *symbolic event*. This event stipulates that the value x was successfully written to the database. $A_{\mathtt{violateRYW}}$ reads as: "this trace *eventually* includes a **writeRsp** message reporting x was successfully written; moreover, *after* this message occurs, there are no further

```
1 assume (x != y) in
2 gen writeReq x in
3 gen writeReq y in
4 let (y1: int) = obs writeRsp in assert (y1 == y) in
5 gen readReq in
6 let (x1: int) = obs writeRsp in assert (x1 == x) in
7 let (y2: int) (s : bool) = obs readRsp in assert (y2 == y && s == true)
```

Fig. 2. A controller P_C that is consistent with $A_{\texttt{violateRYW}}$.

successful writes *until* a **readRsp** message with contents y different from x appears." This formula can be translated into a Symbolic Finite Automata (SFA) [5, 12], on which inclusion and emptiness checks are decidable. Importantly, note that this specification is an *overapproximation* of erroneous traces: not all traces that satisfy this property will be produced by our database model. For example, although the trace **writeRsp**(3); **readRsp**(−1, true) satisfies $A_{\texttt{violateRYW}}$, it does not correspond to a valid execution since it does not contain request the messages that must precede them; these constraints on the expected shape of traces are provided by handler specifications, described below.

Controllers. We introduce a new DSL for expressing controllers that is amenable to automated synthesis. Generating a message in this DSL is analogous to performing an effect in a functional language, with actors playing a similar role to effect handlers [1]. A controller program manages the generation of messages, schedules message order, and constrains data dependencies between messages. Concretely, to realize the trace tr_1, the controller must both issue user-generable messages (e.g., **writeReq**(3)), as well as observable ones that e.g., ensure **writeRsp**(3) is allowed to be delivered before **readRsp**(4, true).

Programs in our DSL are loop-free sequences of commands that generate messages from the environment, and impose constraints on the outputs they observe from the messages sent by the actors under test. Each execution of the program defines a concrete *test*. A program represents a family of such tests because the messages from the environment are only governed by the logical constraints in their specifications: any concrete value consistent with those constraints can be used in a test. We can obtain these values by, e.g., querying a theorem prover. Consequently, new concrete inputs associated with generable messages can lead to new outputs produced by observable ones.

A controller program P_C intended to explore executions that can violate $A_{\texttt{violateRYW}}$ is shown in Fig. 2. Each message is tagged by the keywords **gen** and **obs**, indicating whether it is generated by the user or the database. P_C stipulates an ordering on messages, provides the contents of generable messages, and binds the contents of observable messages to new variables using **let**. The constraints on parameters x and y are defined by the **assume** statement on line 1. Importantly, since the controller does not control the behavior of the actors under test, it cannot mandate the specific values output by the database in message responses. Consequently, assertions may fail; for instance, if the database sends a **readRsp** message with a false status, this would violate the assertion on line 4. Assertions are used to prune executions that will not satisfy $A_{\texttt{violateRYW}}$; our synthesis algorithm adds these assertions selectively (lines 4, 6, and 7) using the PAT specifications associated with each handler. The correctness of this program is established with respect to specifications associated with message handlers that dictate the form and placement of asserts and assumes, as well as the order and structure of **gen** and **obs** statements. We introduce the specification language for handlers below.

2.1 Prophecy Automata Types

In our approach, an actor's behavior is modeled as a set of handler signatures, where a handler's name corresponds to the operation it handles, its parameter types define constraints on message contents, and its return type uses PATS to capture relationships between messages. Absent any

$$\textbf{gen writeReq} : \mathsf{x{:}int} \to [\Box\langle\top\rangle][\mathcal{S}\langle\textbf{writeReq} \mid v = \mathsf{x}\rangle][\Diamond\langle\textbf{writeRsp} \mid v = \mathsf{x}\rangle]$$

$$\textbf{obs writeRsp} : \mathsf{x{:}int} \to [\Box\langle\top\rangle][\mathcal{S}\langle\textbf{writeRsp} \mid v = \mathsf{x}\rangle][\Box\langle\top\rangle]$$

$$\begin{aligned}\textbf{gen readReq} : \mathsf{x{:}int} \dashrightarrow\ & [\Diamond\langle\textbf{writeReq} \mid v = \mathsf{x}\rangle \wedge \neg\bigcirc\Diamond\langle\textbf{writeReq} \mid \top\rangle][\mathcal{S}\langle\textbf{readReq} \mid \top\rangle][\Diamond\langle\textbf{readRsp} \mid v = \mathsf{x} \wedge st = \mathsf{true}\rangle] \\ & \sqcap [\neg\Diamond\langle\textbf{writeReq} \mid \top\rangle][\mathcal{S}\langle\textbf{readReq} \mid \top\rangle][\Diamond\langle\textbf{readRsp} \mid st = \mathsf{false}\rangle]\end{aligned}$$

$$\textbf{obs readRsp} : \mathsf{x{:}int} \to \mathsf{s{:}bool} \to [\Box\langle\top\rangle][\mathcal{S}\langle\textbf{readRsp} \mid v = \mathsf{x} \wedge st = \mathsf{s}\rangle][\Box\langle\top\rangle]$$

Fig. 3. Prophecy Automata Type specifications of message handlers.

expectations about how messages are handled, we cannot prune unrealizable traces when searching for executions that violate a property, e.g., **writeRsp**(3); **readRsp**$(-1, \mathsf{true})$. Doing so requires specifications that constrain every sensible trace in which an actor could be involved; thus, they must be able to capture both temporal properties (e.g., response messages should only follow corresponding request messages) as well as data-dependent ones (e.g., the content of a read response should match the most recent write value). We address this requirement by specifying an actor's message handlers in terms of Pats and use these specifications to compositionally approximate the set of feasible executions. Unlike prior work on trace-based types [21, 28, 47], our formulation accounts for the asynchronous semantics of these systems, where handling one message can trigger the sending of new messages that will only be received later. Intuitively, this means that the return type of a handler include both a “rely" component, specifying assumptions about prior events (the history automaton) that allow this type to be manifested, and a “guarantee" component (the prophecy automaton) that constrains future events.

History, current, and prophecy automata. Pat specifications of the actors in our motivating example are shown in Fig. 3. Return types have the form $[H][\mathcal{S}\langle \mathtt{M} \mid \phi\rangle][F]$, where the three components describe the history, current, and prophecy automata (resp.) that establish the context and effect for any trace containing the message **M**. Each signature reads: “If a message matching $\langle \mathtt{M} \mid \phi\rangle$ appears in a context (trace prefix) accepted by the history automaton H, the future execution (trace suffix) will be accepted by the prophecy automaton F”. Intuitively, prophecy automata are a trace-based analogue of prophecy variables[24] used in other state-based concurrency reasoning approaches to constrain future events. As an example, the first type in Fig. 3 characterizes the behavior of **writeReq** messages. Its history automaton describes how a **writeReq** message is handled in an arbitrary context ($\Box\langle\top\rangle$, where $\Box$ is the globally modality in LTL$_f$), and its prophecy automaton guarantees that a **writeRsp** response message will eventually be issued at some future point, as captured by the $\Diamond$ operator. This specification captures the asynchronous behavior of request/response pairs in our example, requiring that the handler of **writeReq** eventually triggers a **writeRsp** message. On the other hand, we assume little information about the behaviors of the handlers for **readRsp** and **writeRsp** messages, as can been seen by their prophecy automata, which provide no guarantees about any future messages they may produce ($\Box\langle\top\rangle$).

Control flow. A handler's Pat also captures relevant control-flow dependencies. For example, the type of **readReq** uses an intersection type ($\sqcap$) to encode its behaviors in the two different contexts under which a **readReq** message may be handled, corresponding to whether or not some value has been previously written to the database. The first Pat specifies that the handler must eventually respond with the last value that was requested to be written, as captured by the history automaton: $\Diamond\langle\textbf{writeReq} \mid v = \mathsf{x}\rangle \wedge \neg\bigcirc\Diamond\langle\textbf{writeReq} \mid \top\rangle$ and prophecy automaton $\langle\textbf{readRsp} \mid v = \mathsf{x} \wedge st = \mathsf{true}\rangle$. Otherwise, as specified by the second Pat, no value has been successfully written ($\neg\Diamond\langle\textbf{writeRsp} \mid \top\rangle$), and a **readRsp** message with a `false` status will eventually be sent ($\Diamond\langle\textbf{readRsp} \mid st = \mathsf{false}\rangle$).

This specification is sufficiently weak to allow a controller to probe for violations of the RYW property. Specifically, **readReq**'s specification allows a successful **readRsp** to return the value in the database that exists at the time the **readReq** message is handled, ignoring the possibility of other **writeRsp** messages that are executed after the **readReq** but before the corresponding response. This is precisely the scenario depicted by the controller program P_C in Fig. 2 (lines 5-7). On the other hand, a stronger specification for **writeReq** would restrict the controller to focus on executions that exhibit write atomicity, e.g., prohibiting a **readReq** operation from being handled before a **writeRsp**, thus preventing executions that would manifest a RYW violation:

$$\textbf{gen}\ \mathtt{\textbf{writeReq}}\ : \mathtt{x{:}int} \to [\Box\langle\top\rangle][\mathcal{S}\langle\mathtt{\textbf{writeReq}} \mid v = \mathtt{x}\rangle][(\neg\langle\mathtt{\textbf{readReq}} \mid \top\rangle)\ \mathcal{U}\ \langle\mathtt{\textbf{writeRsp}} \mid v = \mathtt{x}\rangle]$$

PATs thus provide an expressive framework in which to specify the set of executions that are of interest to the test engineer, grounded in the semantic relationships that are expected to hold among different actors in the model: weaker specifications admit more behaviors, at the potential cost of trying to explore executions that are not realizable by the actors' implementations; stronger specifications restrict this set, at the cost of excluding some potentially erroneous executions.

Typechecking. Specifying the behavior of actors in terms of PATs allows us to use a type system to statically check that controller programs will focus on realizable executions, i.e.,

Well-typed controller programs do not generate uninteresting traces

For example, to type the use of **readReq** on line 5 in Fig. 2, we first "divide" P_C into three pieces: a history (line 1 - 4), an action (line 5), and a future (line 6 - 7). As P_C encodes a family of executions, the first subprogram corresponds to the set of contexts that can occur before **readReq** is handled, while the last subprogram captures all the traces that may follow. Thus, we must ensure that each of these pieces are consistent with the type of **readReq**, which requires that the last value written to the database is y (line 3) in the history, that the message being handled is **readReq**, and that a **readReq** message with value y will be produced in the future (line 7). Notably, P_C can indeed induce a trace that violates RYW consistency. We can show this by typechecking P_C against the PAT $[\Box\langle\bot\rangle][A_{\mathtt{violateRYW}}][\Box\langle\bot\rangle]$. This PAT asserts that when there are no prior messages ($\Box\langle\bot\rangle$), the execution of the controller generates a trace consistent with $A_{\mathtt{violateRYW}}$, and no more future messages are generated ($\Box\langle\bot\rangle$).

2.2 Controller Synthesis

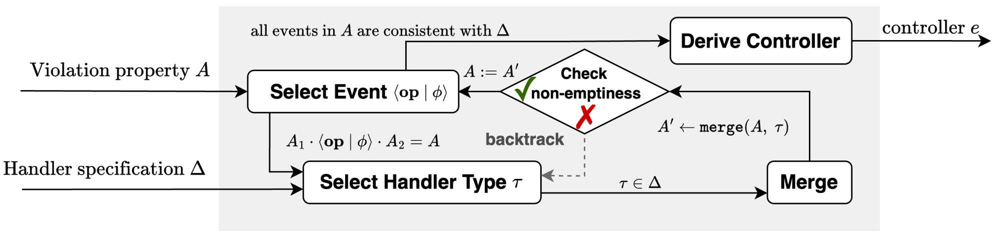
all events in A are consistent with Δ
Derive Controller
controller e
Violation property A
Select Event $\langle op \mid \phi\rangle$
$A := A'$
Check non-emptiness
$A' \leftarrow \mathtt{merge}(A, \tau)$
$A_1 \cdot \langle op \mid \phi\rangle \cdot A_2 = A$
backtrack
Handler specification Δ
Select Handler Type τ
$\tau \in \Delta$
Merge

Fig. 4. Test controller synthesis pipeline.

Interpreting messages as effects allows us to frame the derivation of a controller as a component-based synthesis problem, guided by the PAT specifications of the actors comprising the SUT. Fig. 4 gives a high-level overview of our algorithm, which consists of two phases. In the first phase, we systematically refine an automaton that captures violations of our target property A to remove traces that do not correspond to feasible executions. The resulting automaton A' encodes a stronger property on traces, i.e., $A' \subseteq A$, which ensures that each message is consistent with its specification. In the second phase, we use A' to derive a controller program. As an example, the set of traces

$$
\begin{array}{rrl}
\textbf{Variables} & & x, y, z, \nu, \ldots \\
\textbf{Base Types} & b ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{int} \mid \ldots \\
\textbf{Pure Operations} & op ::= & + \mid - \mid == \mid < \mid \leq \mid \ldots \\
\textbf{Constants} & c ::= & () \mid \mathbb{B} \mid \mathbb{Z} \mid \ldots \\
\textbf{Values} & v ::= & c \mid x \\
\textbf{Qualifiers} & \phi ::= & v \mid op\ \overline{v} \mid \bot \mid \top \mid \neg\phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \implies \phi \mid \forall x{:}b.\, \phi \\
\textbf{Effectful Operations} & \mathbf{op} ::= & \mathbf{readReq} \mid \mathbf{readRsp} \mid \ldots \\
\textbf{Message Kinds} & k ::= & \mathbf{gen} \mid \mathbf{obs} \\
\textbf{Expressions} & e ::= & v \mid \mathbf{let}\ x{:}b = op\ \overline{v}\ \mathbf{in}\ e \mid \mathbf{gen}\ \mathbf{op}\ \overline{v}\ \mathbf{in}\ e \mid \mathbf{let}\ \overline{x{:}b} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e \\
& & \mid \mathbf{assert}\ \phi\ \mathbf{in}\ e \mid \mathbf{assume}\ \phi\ \mathbf{in}\ e \mid e \oplus e
\end{array}
$$

Fig. 5. λ^C syntax

captured by $A_{\mathtt{violateRYW}}$ can be refined into:

$$
\begin{aligned}
&\mathcal{S}\langle \mathbf{writeReq} \mid v = \mathsf{x}\rangle \cdot \mathcal{S}\langle \mathbf{writeReq} \mid v = \mathsf{y} \land v \neq \mathsf{x}\rangle \cdot \mathcal{S}\langle \mathbf{writeRsp} \mid v = \mathsf{y}\rangle \cdot \\
&\mathcal{S}\langle \mathbf{readReq} \mid \top\rangle \cdot \mathcal{S}\langle \mathbf{writeRsp} \mid v = \mathsf{x}\rangle \cdot \mathcal{S}\langle \mathbf{readRsp} \mid v = \mathsf{y} \land st = \mathsf{true} \land v \neq \mathsf{x}\rangle
\end{aligned}
\qquad (A'_{\mathtt{violateRYW}})
$$

This automaton specializes the set of traces captured by the subformula under the $\Diamond$ operator in $A_{\mathtt{violateRYW}}$, stipulating specific instances of **writeReq** and **readReq** events, in a way that is consistent with the specification of their handlers. The final two events in $A'_{\mathtt{violateRYW}}$ align with the first and last events in $A_{\mathtt{violateRYW}}$, ensuring all traces satisfying $A'_{\mathtt{violateRYW}}$ also satisfy $A_{\mathtt{violateRYW}}$. Observe that the structure of $A'_{\mathtt{violateRYW}}$ closely resembles the controller program P_C, with the main difference being that P_C is more operational, dividing messages into **gen** (generable) and **obs** (observable) groups. The controller also provides the contents for **gen** messages, while the contents of **obs** messages are constrained only by local variables.

Property Refinement Loop. The refinement loop is a crucial piece of the algorithm in Fig. 4, as it ensures that the traces accepted by the refined formula are consistent with our expectations of handler behaviors. Viewed from another perspective, this algorithm searches for a family of traces that witness a property violation until one is found that aligns with the provided specifications. A key challenge is dealing with temporal modalities, like $\Diamond$ and $\mathcal{U}$, that permit an arbitrary number of possible messages in the trace before the messages of interest occur. $A'_{\mathtt{violateRYW}}$, for example, includes six events, not all of which appear in $A_{\mathtt{violateRYW}}$ (e.g., **readReq**). While these modalities allow us to refine the current formula by adding new messages, each of these messages can impose new requirements that must be satisfied. To address this challenge, our algorithm lazily injects new messages in the controller program, and then recursively repairs any unmet obligations. As an example, when working on a **readRsp** message, the last message in $A_{\mathtt{violateRYW}}$, the algorithm identifies that it must have been issued by the handler for **readReq** (via the first case of the intersection type in its Pat). Moreover, **readReq**'s type also indicates that its content y should belong to a previous **writeReq**. Based on these constraints, our algorithm refines the current formula by adding **writeReq** and **readReq** messages before **readRsp**, and marks both as messages whose constraints still need to be satisfied as synthesis proceeds.

3 Formalization

We formalize our approach using a core language, λ^C, for expressing controller programs. This language is a call-by-value and asynchronous message-passing calculus that abstracts away the implementation details of the actors that the controller interacts with, focusing only on the structure of the controller program itself. The syntax of λ^C is shown in Fig. 5; it includes both pure and effectful operations (*op* and **op**), non-deterministic choice ($\oplus$), and assertions. Effectful operations are categorized as either *generable* (**gen**) or *observable* (**obs**).

$$\textbf{Messages} \quad m ::= \mathbf{op}(\overline{c}) \qquad \textbf{Buffers} \quad \beta \in \mathcal{P}(m) \qquad \textbf{Traces} \quad \alpha ::= [\,] \mid m :: \alpha \mid \alpha +\!\!+ \alpha$$

$$\textbf{Handler Semantics} \; \boxed{\alpha \vDash \mathbf{op}(\overline{c}) \Downarrow \beta} \qquad \textbf{Operational Semantics} \; \boxed{\phi \Downarrow c \quad \alpha \vDash (\beta, e) \xhookrightarrow{\alpha} (\beta, e)}$$

$$\frac{\beta = \{\mathbf{op}(\overline{c})\} \cup \beta_1 \quad \alpha \vDash \mathbf{op}(\overline{c}) \Downarrow \beta_2 \quad e' = e[\overline{x \mapsto c}]}{\alpha \vDash (\beta, \mathbf{let}\ \overline{x} = \mathbf{obs\ op\ in}\ e) \xhookrightarrow{[\mathbf{op}(\overline{c})]} (\beta_1 \cup \beta_2, e')} \textsc{StObs} \qquad \frac{\alpha \vDash \mathbf{op}(\overline{c}) \Downarrow \beta'}{\alpha \vDash (\beta, \mathbf{gen\ op}\ \overline{c}\ \mathbf{in}\ e) \xhookrightarrow{[\mathbf{op}(\overline{c})]} (\beta \cup \beta', e)} \textsc{StGen}$$

$$\frac{\phi[\overline{x \mapsto c}] \Downarrow \top}{\alpha \vDash (\beta, \mathbf{assume}\ \phi\ \mathbf{in}\ e) \xhookrightarrow{[\,]} (\beta, e[\overline{x \mapsto c}])} \textsc{StAssume} \qquad \frac{\phi \Downarrow \top}{\alpha \vDash (\beta, \mathbf{assert}\ \phi\ \mathbf{in}\ e \xhookrightarrow{[\,]} (\beta, e)} \textsc{StAssert}$$

Fig. 6. Selected Operational Semantics

Operational Semantics. Messages in $\boldsymbol{\lambda}^C$ are operations applied to concrete values ($\mathbf{op}(\overline{c})$). Evaluating a $\boldsymbol{\lambda}^C$ program depends on an input *trace*, i.e., a sequence of messages, and an input *buffer*, i.e., an element of a multiset of messages. Each evaluation step produces an output trace and an updated buffer. Traces are equipped with the standard list operations (i.e., cons :: and concatenation $+\!\!+$). The operational semantics of $\boldsymbol{\lambda}^C$ are defined by the small-step reduction relation: $\alpha \vDash (\beta, e) \xhookrightarrow{\alpha'} (\beta', e')$. This judgment is read as: "under the context α and current message buffer β, e steps to e', emitting the trace α' and producing the output buffer β'." Intuitively, the context α represents the sequence of messages visible to a handler, thereby determining its response; the buffer β contains messages that have been issued but not yet been made visible to a handler. The semantics uses an auxiliary judgement, $\alpha \vDash \mathbf{op}(\overline{c}) \Downarrow \beta$, that specifies any new messages that need to be added to the message buffer after handling **op**.

Fig. 6 provides the key rules of $\boldsymbol{\lambda}^C$'s semantics.[2] The rule for observable events (StObs) reflects the "receive-and-send" behavior of messages produced by handlers. This rule non-deterministically removes a pending message that matches the effectful operation **op**, evaluates it under the current context, and substitutes the message payload $\overline{c}$ for the variables $\overline{x}$ in e, the body of the **let** expression. Any new messages generated as a consequence of handling **op** $\overline{c}$ are added to the resulting message buffer. The reduction rule for generable events (StGen) is similar, but since these events can be directly performed by the controller, the rule does not require a corresponding message in the buffer. The StAssume rule substitutes the variables $\overline{x}$ with values $\overline{c}$ that satisfy the qualifier ϕ in the body of an assume expression. The StAssert rule, in contrast, requires the qualifier of an assert expression to hold in order for it to make progress.

Example 3.1 (Operational Semantics). The first three events in the trace tr_1 are produced by the controller program $A_{\texttt{violateRYW}}$ as follows:

$$\begin{array}{lr}
[\,] \vDash (\emptyset, P_C) \xhookrightarrow{[\,]} (\emptyset, \textit{lines } 2 - 7 \textit{ of } P_C, \textit{with } x \mapsto 3, y \mapsto 4) & (\textsc{StAssume}) \\
\xhookrightarrow{[\mathtt{writeReq}(3)]} (\{\mathtt{writeRsp}(3)\}, \textit{lines } 3 - 7 \textit{ of } P_C) & (\textsc{StGen}) \\
\xhookrightarrow{[\mathtt{writeReq}(4)]} (\{\mathtt{writeRsp}(3), \mathtt{writeRsp}(4)\}, \textit{lines } 4 - 7 \textit{ of } P_C) & (\textsc{StGen}) \\
\xhookrightarrow{[\mathtt{writeRsp}(4)]} (\{\mathtt{writeRsp}(3)\}, \textit{lines } 5 - 7 \textit{ of } P_C) & (\textsc{StObs}, \textsc{StAssert}) \\
\xhookrightarrow{[\mathtt{readReq}]} (\{\mathtt{writeRsp}(3), \mathtt{readRsp}(4, \mathtt{true})\}, \textit{lines } 6 - 7 \textit{ of } P_C) & (\textsc{StGen})
\end{array}$$

The first step performs the substitution $(x \mapsto 3, y \mapsto 4)$, which satisfies the assumed formula $x \neq y$ (line 1). In the next two steps, P_C generates two **writeReq** messages and adds two **writeRsp** messages to the message buffer. One of these messages is consumed by the fourth step, causing the assertion on line 4 of Fig. 2 to succeed. The fifth step handles **readReq**, and the message **readRsp(4, true)** is added to the buffer.

[2]The remaining rules are completely standard and provided in the supplemental material.

$$
\begin{array}{rrcl}
\textbf{Pure Refinement Types} & t & ::= & \{v{:}b \mid \phi\} \mid x{:}t \to t \\
\textbf{Symbolic LTL}_f & H, A, F & ::= & \langle \mathbf{op}\ \overline{x} \mid \phi \rangle \mid \langle \phi \rangle \mid \neg A \mid A \land A \mid A \lor A \mid A \cdot A \mid \bigcirc A \mid A\, \mathcal{U}\, A \\
\textbf{Prophecy Automata Types} & \tau & ::= & [H][A][F] \mid x{:}b \dashrightarrow \tau \mid x{:}t \to \tau \mid \tau \sqcap \tau \\
\textbf{Type Contexts} & \Gamma & ::= & \emptyset \mid x{:}t, \Gamma
\end{array}
$$

Fig. 7. λ^U types.

3.1 Types

The syntax of types in λ^C is shown in Fig. 7. Types include *pure* refinement types, which describe pure computations, and Prophecy Automata Types (Pats), which describe effectful computations. Pure refinement types are similar to those found in other refinement type systems [19], and allow base types (e.g., `int`) to be further constrained by a logical formula or qualifier. Verification conditions generated by our type-checker can be encoded as effectively propositional (EPR) sentences [37], which can be efficiently handled by an off-the-shelf theorem prover such as Z3 [7].

Symbolic Finite Automata. Following other recent trace-based type systems[47], λ^C uses *Symbolic Finite Automata* (SFAs) [5, 12, 43] to qualify traces, similar to how standard refinement types use formulae to qualify the types of pure terms. We use a symbolic version of LTL$_f$ to express SFAs. A *symbolic event* $\langle \mathbf{op}\ \overline{x} \mid \phi \rangle$ is an atomic predicate that describes an effectful operation **op** whose inputs $\overline{x}$ must satisfy the qualifier ϕ.[3] The standard temporal operators (e.g., test $\langle \phi \rangle$, next $\bigcirc A$, until $\mathcal{U}$) and various set operators (i.e., complement $\neg$, intersection $\land$, and union $\lor$) are defined normally. These operators are sufficient to capture other modalities, e.g., eventually ($\Diamond$), globally ($\Box$), and importantly, the singleton (last) modality $\mathcal{S}$, which describes a singleton trace, i.e., one which prohibits any subsequent effects [6]. SFAs can capture several common patterns: the set of all possible traces $\Box\langle\top\rangle$, the singleton set containing the empty trace $\Box\langle\bot\rangle$, and the empty set of traces and $\neg\Box\langle\top\rangle$; these are analogous to the regular expressions $.^*$, ϵ, and $\emptyset$, resp.

Prophecy Automata Types. A Pat $[H][A][F]$ is comprised of three SFAs: a *history* SFA H that captures the context traces (i.e., a sequence of visible, already handled, symbolic events) in which a term can be executed, a *current* SFA A that describes newly handled messages that arise from executing a term, and a *prophecy* SFA F that characterizes new messages that have yet to be performed. Function types use Pats in their result types to describe the effects they perform, when combined with intersection types ($\sqcap$), this allows users to express complex control flows. Function types also use *ghost variables* ($x{:}b \dashrightarrow \tau$) to capture data dependencies among symbolic events; for example, the full signature of the **getReq** handler from Sec. 1 uses the ghost variables `key` and `val`.

Example 3.2 (Strong Consistency). Strong consistency requires that all **getRsp** messages report the last value that was **put** to the database. This property is captured by the following Pat:

$$
\begin{array}{l}
\texttt{val}{:}\texttt{tVal} \dashrightarrow \texttt{key}{:}\{v{:}\texttt{tKey} \mid \top\} \to [\Diamond\langle \mathbf{putReq} \mid k = \texttt{key} \land v = \texttt{val}\rangle \land \bigcirc\neg\Diamond\langle \mathbf{putReq} \mid k = \texttt{key}\rangle] \\
\qquad\qquad [\mathcal{S}\langle \mathbf{getReq} \mid k = \texttt{key}\rangle][(\neg\langle \mathbf{putReq} \mid k = \texttt{key}\rangle)\ \mathcal{U}\ \langle \mathbf{getRsp} \mid k = \texttt{key} \land v = \texttt{val}\rangle]
\end{array}
$$

The prophecy automata in this Pat requires that no updates (**putReq**) to `key` in the database happen before a user receives a response to a **getReq** message for the key `key`.

3.2 Typing rules

Our typing judgment features three contexts: a type context Γ, a handler context Δ, and a capability context Θ. The type context, Γ maps from variables to pure refinement types (i.e., t). As

[3]When the fields of an event are clear from context, we omit its parameters $\overline{x}$, e.g., $\langle \mathbf{writeReq} \mid v > 0\rangle$ means $\langle \mathbf{writeReq}\ v \mid v > 0\rangle$.

Auxiliary Typing $\boxed{\Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma \vdash A \subseteq A \quad \Gamma \vdash \tau <: \tau}$ **Typing** $\boxed{\Gamma \vdash v : t \quad \Gamma;\ \Delta;\ \Theta \vdash e : \tau}$

$$\frac{\Gamma \vdash^{\mathbf{WF}} H \quad \Gamma \vdash^{\mathbf{WF}} A \quad \Gamma \vdash^{\mathbf{WF}} F \quad \Gamma \vdash H{\cdot}A{\cdot}F \not\subseteq \neg\Box\langle\top\rangle}{\Gamma \vdash^{\mathbf{WF}} [H][A][F]}\ \textsc{WfHAF} \qquad \frac{\Gamma \vdash H_2 \subseteq H_1 \quad \Gamma \vdash A_1 \subseteq A_2 \quad \Gamma \vdash F_1 \subseteq F_2}{\Gamma \vdash [H_1][A_1][F_1] <: [H_2][A_2][F_2]}\ \textsc{SubHAF} \qquad \frac{\Gamma;\Delta;\Theta \vdash e : \tau \quad \Gamma;\Delta;\Theta \vdash \tau <: \tau'}{\Gamma;\Delta;\Theta \vdash e : \tau'}\ \textsc{TSub}$$

$$\frac{\Gamma;\Delta;\Theta \vdash e_1 : \tau \quad \Gamma;\Delta;\Theta \vdash e_2 : \tau}{\Gamma;\Delta;\Theta \vdash e_1 \oplus e_2 : \tau}\ \textsc{TChoice} \qquad \frac{\Delta(\mathbf{op}) = \langle \mathbf{gen}\ \tau, \Theta' \rangle \quad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \rightarrow [H][\mathcal{S}\langle \mathbf{op} \mid \phi \rangle][A{\cdot}F] \quad \forall i.\Gamma \vdash v_i : t_i \quad \Gamma;\Delta;\Theta \cup \Theta' \vdash e : [H{\cdot}\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle][A][F]}{\Gamma;\Delta;\Theta \vdash \mathbf{gen}\ \mathbf{op}\ \overline{v_i}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle{\cdot}A][F]}\ \textsc{TGen}$$

$$\frac{}{\Gamma;\Delta;\emptyset \vdash () : [H][\Box\langle\bot\rangle][F]}\ \textsc{TRet} \qquad \frac{\Delta(\mathbf{op}) = \langle \mathbf{obs}\ \tau, \Theta' \rangle \quad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \rightarrow [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \rangle][A{\cdot}F] \quad \Gamma, \overline{x{:}t};\Delta;\Theta \cup \Theta' \vdash e : [H{\cdot}\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \wedge \overline{y = x}\rangle][A][F]}{\Gamma;\Delta;\{\mathbf{op}\} \cup \Theta \vdash \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \rangle{\cdot}A][F]}\ \textsc{TObs}$$

Fig. 8. Selected typing rules.

in other trace-based refinement type systems, contexts are not allowed to contain Pats— doing so breaks several structural properties (e.g., weakening) that are used to prove type safety. The handler context, Δ, maps operations to two key pieces of information: a specification of its handler as a Pat that is tagged with whether it is observable or generable, and the operations its handler adds to the buffer. The capability context, Θ, records the set of observable messages that are in scope. This context is used to ensure that every observation corresponds to a message that was triggered by a previous event.

Example 3.3. The handler context Δ for our running examples augments the four specifications from Fig. 3 as follows:

$$\Delta \doteq \{(\mathbf{readReq}, \langle \ldots, \{\mathbf{readRsp}\}\rangle), (\mathbf{readRsp}, \langle \ldots, \emptyset\rangle), (\mathbf{writeReq}, \langle \ldots, \{\mathbf{writeRsp}\}\rangle), (\mathbf{writeRsp}, \langle \ldots, \emptyset\rangle)\}$$

Auxiliary typing relations. Our system depends on three auxiliary relations: a well-formedness relation $\Gamma \vdash^{\mathbf{WF}} \tau$ which ensures, e.g., that all qualifiers appearing in a type τ are closed under the current typing context Γ; an inclusion relation on SFAs $\Gamma \vdash A \subseteq A$; and a mostly-standard semantic subtyping relation. Fig. 8 provides two of the key rules for these relations. A well-formed Pat (WfHAF) is required to accept at least one trace ($\neg\Box\langle\top\rangle$ is an SFA that rejects all traces). Subtyping for two Pats (SubHAF) is established by checking inclusion between their constituent automata under the current type context Γ. Inclusion on the history and prophecy automata is contravariant, while current automata are covariant. Intuitively, since both the history and prophecy automata restrict the contexts in which a term that produces the current automata may appear, it is safe to further constrain both contexts.

Typing Rules. A subset of our typing rules is shown in Fig. 8.[4] All of our terms assume any types they use are well-formed, so we elide the corresponding well-formedness judgments from their premises. The rules for performing events, TGen and TObs, both extract the type of the corresponding handler from Δ, $[H][\mathcal{S}\langle \mathbf{op} \mid \phi \rangle][A{\cdot}F]$, and require that it aligns with the Pat of the expression that the operation is being performed in:

$$\underbrace{H}_{\text{history}} \cdot \underbrace{\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle{\cdot}A}_{\text{current}} \cdot \underbrace{F}_{\text{prophecy}} \equiv \underbrace{H}_{\text{history}} \cdot \underbrace{\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle}_{\text{current}} \cdot \underbrace{A{\cdot}F}_{\text{prophecy}}$$

To type the rest of the expression, both rules move the symbolic event $\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle$ from the head of the current automata to the tail of the history automata and add any new capabilities to Θ. In order to make an observation on **op**, TObs additionally requires that the capability context has a corresponding capability ($\{\mathbf{op}\} \cup \Theta$). The standard subsumption rule TSub allows us to change the

[4]The complete set of typing rules is included in the supplemental material.

shape of a type that a term is being typed against. Controllers always end in a unit value (); thus, the TRet rule requires the current automata of this term ($\Box\langle\bot\rangle$) to only accept the empty trace (i.e., []). The nondeterministic choice operator is typed using the TChoice rule, when combined with the subsumption rule, this allows controllers to explore different message orderings.

Example 3.4 (Controller Typing). We provide an informal typing derivation of P_C against a Pat that encodes a violation of an RYR policy, $[\Box\langle\bot\rangle][A_{\texttt{violateRYW}}][\Box\langle\bot\rangle]$, under the type context $\Gamma \doteq \texttt{x}{:}\{v{:}\texttt{int} \mid \top\}, \texttt{y}{:}\{v{:}\texttt{int} \mid v \neq x\}$. The first step of our derivation uses TSub to refine our target type to a Pat that better aligns with the messages sent by P_C:

$$A'_{\texttt{violateRYW}} \doteq \underbrace{\mathcal{S}\langle\textbf{writeReq} \mid v = \texttt{x}\rangle}_{A_1} \cdot \underbrace{\mathcal{S}\langle\textbf{writeReq} \mid v = \texttt{y} \land v \neq \texttt{x}\rangle\cdot\mathcal{S}\langle\textbf{writeRsp} \mid v = \texttt{y}\rangle\cdot\mathcal{S}\langle\textbf{readReq} \mid \top\rangle}_{A_2} \cdot$$
$$\underbrace{\mathcal{S}\langle\textbf{writeRsp} \mid v = \texttt{x}\rangle}_{A_3} \cdot \underbrace{\mathcal{S}\langle\textbf{readRsp} \mid v = \texttt{y} \land st = \texttt{true} \land v \neq \texttt{x}\rangle}_{A_4}$$

The first **gen** expression on line 2 of P_C is then typed using TGen. After retrieving the specification of **writeReq** from Δ and uses TSub to adjust it into a shape consistent with $A'_{\texttt{violateRYW}}$:

$$\frac{\begin{array}{l}\Delta(\textbf{writeReq}) = \textbf{gen}\ \langle \texttt{x}{:}\texttt{int} \to [\Box\langle\top\rangle][\mathcal{S}\langle\textbf{writeReq} \mid v = \texttt{x}\rangle][\Diamond\langle\textbf{writeRsp} \mid v = \texttt{x}\rangle], \{\textbf{writeRsp}\}\rangle \\ \Gamma \vdash \texttt{x}{:}\texttt{int} \to [\Box\langle\top\rangle][\mathcal{S}\langle\textbf{writeReq} \mid v = \texttt{x}\rangle][\Diamond\langle\textbf{writeRsp} \mid v = \texttt{x}\rangle] \\ \quad <: \texttt{x}{:}\texttt{int} \to [\Box\langle\bot\rangle]\ [\mathcal{S}\langle\textbf{writeReq} \mid v = \texttt{x}\rangle]\ [A_2\cdot\mathcal{S}\langle\textbf{writeRsp} \mid v = \texttt{x}\rangle\cdot A_4]\end{array}}{\Gamma; \Delta; \emptyset \vdash P_C \text{ (lines 2 - 7)} : [\Box\langle\bot\rangle][A_1\cdot A_2\cdot A_3\cdot A_4][\Box\langle\bot\rangle]} \quad (\textsc{TGen})$$

Since the type of **writeReq** aligns with the target type $[\Box\langle\bot\rangle][A_1\cdot A_2\cdot A_3\cdot A_4][\Box\langle\bot\rangle]$, we continue typing the rest of P_C (lines 3 - 7) against the Pat $[A_1][A_2\cdot A_3\cdot A_4][\Box\langle\bot\rangle]$.

3.3 Type Soundness

Type denotations. Similar to other refinement type systems [19], types in $\boldsymbol{\lambda}^C$ are denoted as their inhabitants (i.e., $[\![t]\!]$ and $[\![\tau]\!]$). The capability context is denoted as message buffers, while the type context Γ is denoted as *substitution* σ (e.g., $[x \mapsto 3, y \mapsto 4]$ in Theorem 3.1) that provides the assignments for binding variables in Γ. Moreover, the denotation (accepting language) of SFAs is the set of traces they can accept. Then, automata inclusion under a type context is defined as $\Gamma \vdash A \subseteq A' \doteq \forall\sigma \in [\![\Gamma]\!].[\![\sigma(A)]\!] \subseteq [\![\sigma(B)]\!]$.[5]

Well-formed Handler specification. A handler specification Δ should be consistent with the auxiliary semantics of handlers introduced in Fig. 6, also, Δ should also guarantee the new sending message assumed by capability context can be eventually received.

Definition 3.5 (Well-formed handler context). The handler specification Δ is well-formed iff for all operator **op** and its Pat $\overline{y{:}b} \dashrightarrow \overline{x{:}t} \to [H][\mathcal{S}\langle\textbf{op}\ \overline{y} \mid \phi\rangle][F]$ and capability $\{\overline{\textbf{op}_\textbf{i}}\}$ in Δ satisfying

$$\forall\overline{y{:}b}.\forall\alpha_h \in [\![H]\!].\forall\overline{c \in [\![t]\!]}.\forall\overline{c_{ij}}.\forall\overline{\alpha_i}.\alpha_1 +\!\!\!+ [\textbf{op}_\textbf{1}(\overline{c_{1j}})] +\!\!\!+ \ldots[\textbf{op}_\textbf{n}(\overline{c_{nj}})] +\!\!\!+ \alpha_{n+1} \in [\![F]\!] \Longrightarrow$$
$$\alpha_h \vDash \textbf{op}(\overline{c}) \Downarrow \{\textbf{op}_\textbf{i}(\overline{c_{ij}})\} \land \phi[\overline{x \mapsto c}]$$

Theorem 3.6 (Fundamental Theorem). *A well-typed term, i.e.,* $\Gamma; \Delta; \Theta \vdash e : [H][A][F]$, *generates traces consistent with the* Pat *and can also terminate with a message buffer denoted by capability* Θ.[6]

$$\forall\sigma \in [\![\Gamma]\!].\sigma(e) \in [\![\sigma([H][A][F])]\!] \land \forall\alpha_h \in [\![\sigma(H)]\!].\forall\beta \in [\![\Theta]\!].\exists\alpha.\exists\beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ())$$

Theorem 3.7 (Type Soundness). *Given a well-formed handler specification* Δ, *with ghost variables* $\overline{x{:}b}$ *and a violation property* A, *a controller* e *that satisfies* $\overline{x}{:}\ \{v{:}b \mid \top\}; \Delta; \emptyset \vdash e : [\Box\langle\bot\rangle][A][\Box\langle\bot\rangle]$ *will realize at least one trace consistent with* A, *i.e.,*

$$\exists\overline{c{:}b}.\exists\alpha.[\,] \vDash (\emptyset, e[\overline{x \mapsto c}]) \overset{\alpha}{\hookrightarrow}{}^{*} (\emptyset, ()) \land \alpha \in [\![A[\overline{x \mapsto c}]]\!]$$

[5]The details of these definition can be found in our supplemental material.
[6]The proofs of all theorems in this paper are provided in the supplemental material.

4 Synthesis

When typing a program using our declarative typing rules, we can freely apply the subsumption rule to align the (ordered) set of events performed by the program with a Pat that describes a user's desired high-level property. Any *synthesis* procedure based on such a high-level specification must devise a similar ordering alongside the events in the program it generates. At the same time, each event needs to align with the specification of its handler in Δ, i.e., its temporal and data-dependency constraints must be satisfied. Our solution to this problem is a refinement loop, depicted in Fig. 4, that iteratively refines the high-level specification into one that is consistent with these constraints. Each iteration of this loop targets a single event, adding events before and after that message so that its dependences are satisfied, i.e., so that the corresponding handler at that point in the synthesized program is well-typed. While declarative typing rules always assume Pats are well-formed, our loop employs an automata non-emptiness check to ensure it represents a controller that produces at least one feasible trace. After the refinement loop has finished, a corresponding well-typed controller program can be mechanically extracted from the refined property.

4.1 Abstract trace

Our algorithm targets automata that have been normalized into an *abstract trace*, a sequence of singleton events $\mathcal{S}\langle \textbf{op} \mid \phi \rangle$. This normal form makes it easy to identify the traces that must precede and follow each event $\langle \textbf{op} \mid \phi \rangle$ in an SFA's traces.

Definition 4.1 (Abstract Trace). An abstract trace Π is an SFA, encoded by a symbolic LTL$_f$ formula defined by the following grammar:

$$\textbf{Abstract Trace} \quad \Pi ::= \quad \mathcal{S}\langle \textbf{op}\ \overline{x} \mid \phi \rangle \mid \Box A \mid \Pi \cdot \Pi$$

Every symbolic LTL$_f$ formula can be normalized into a finite set of abstract traces.

Example 4.2 (Abstract trace). The formula encoding violations of a Read-Your-Writes policy, $A_{\texttt{violateRYW}}$, can be normalized into the following abstract trace:

$$\mathcal{S}\langle \texttt{writeRsp} \mid v = \mathsf{x} \rangle \cdot \Box(\neg\langle \texttt{writeRsp} \mid \top \rangle) \cdot \mathcal{S}\langle \texttt{readRsp} \mid v = \mathsf{y} \wedge st = \mathsf{true} \wedge \mathsf{y} \neq \mathsf{x} \rangle \cdot \Box\langle \top \rangle \qquad (\Pi_{\texttt{violateRYW}})$$

This formula captures the executions of our database example in which a successful **readRsp** message carries a value different from the last observed **writeRsp** message.

4.2 Synthesis Algorithm

Our top-level synthesis algorithm is shown in Algorithm 1. Given an (unsafe) abstract trace Π and corresponding ghost variables (e.g., x and y in $\Pi_{\texttt{violateRYW}}$) as input, this nondeterministic algorithm synthesizes a well-typed $\boldsymbol{\lambda}^C$ controller. The algorithm follows the structure given in Fig. 4, using a refinement loop (lines 3 - 9) to refine the input abstract trace Π and then deriving[7] the final controller from the refined property (line 10). Each iteration of this loop nondeterministically chooses a target event that is used to refine the current abstract trace; different choices may result in different message orders, and some of these choices may cause the algorithm to fail. Our implementation resolves this nondeterminism in the algorithm via an efficient backtracking search procedure that takes the union of all successful runs in order to capture different orderings.

Event dependencies. The refined abstract trace produced by our loop must correspond to a well-typed program, i.e., the traces preceding and following each of its events must be consistent with the specifications of its corresponding handler. The events that will precede and follow each event are not known until the loop is finished, so we cannot simply track the set of observable events, as the declarative typing rules did via Θ. Instead, each iteration of the loop detects the

[7]The definition of both the SFA normalization procedure and **DeriveTerm** are provided in the supplemental material.

Algorithm 1: Synthesis

```
Inputs  : Ghost variables $\overline{x{:}b}$, handler context $\Delta$, and abstract unsafe trace $\Pi$
Output  : Controller $e$, such that $\Gamma;\Delta;\emptyset \vdash e \;:\; [\Box\langle\bot\rangle][\Pi][\Box\langle\bot\rangle]$
1  $\Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}} \leftarrow \emptyset, \emptyset$ ; // initialize sets of forward and backward dependencies.
2  $\Gamma \leftarrow \overline{x{:}\{v{:}b \mid \top\}}$ ; // initialize type context
   // Pick a previously unexamined symbolic event ⟨op | φ⟩
3  while exists $\langle \mathbf{op}\ \overline{x} \mid \phi\rangle$ s.t. $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \mathbf{op}\ \overline{x} \mid \phi\rangle{\cdot}\Pi_f$ and $\langle \mathbf{op}\ \overline{x} \mid \phi\rangle \notin (\Theta_{\mathsf{fw}} \cap \Theta_{\mathsf{bw}})$ do
4      if $\mathbf{op} \notin \Theta_{\mathsf{fw}}$ then
5          $(\Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op}(\overline{y})\rangle, \Pi_f) \leftarrow$ Forward$(\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op}(\overline{y})\rangle, \Pi_f)$;
6      if $\mathbf{op} \notin \Theta_{\mathsf{bw}}$ then
7          if Generable(op) then $\Theta_{\mathsf{bw}} \leftarrow \Theta_{\mathsf{bw}} \cup \{\mathbf{op}\}$ ;
8          else $(\Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op}(\overline{y})\rangle, \Pi_f) \leftarrow$ Backward$(\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op}(\overline{y})\rangle, \Pi_f)$ ;
9      $\Pi \leftarrow \Pi_h{\cdot}\mathcal{S}\langle \mathbf{op}(\overline{y})\rangle{\cdot}\Pi_f$ ; // refine unsafe abstract trace
10 $e \leftarrow$ DeriveTerm$(\Gamma, \Pi)$ ; // derive controller program
11 return $e$;
```

unresolved dependencies of a target event in the current abstract trace and repairs them by inserting appropriate events before and after it. Intuitively, each observable event $\mathbf{op}_{\mathtt{child}}$ must follow an operation $\mathbf{op}_{\mathtt{parent}}$, whose handler produces it, forming a tree-like dependency structure similar to that described by Mora et al. [27]. The refinement loop reconstructs these trees from the target node, refining the candidate abstract trace into one that satisfies these dependencies.

Example 4.3 (Message dependency). The refined unsafe abstract trace $A'_{\mathtt{violateRYW}}$ contains three distinct sets of messages, comprised of pairs of requests and corresponding responses:

$$\mathcal{S}\langle \mathbf{writeReq} \mid v = \mathsf{x}\rangle{\cdot}\mathcal{S}\langle \mathbf{writeReq} \mid v = \mathsf{y} \land v \neq \mathsf{x}\rangle\mathcal{S}\langle \mathbf{writeRsp} \mid v = \mathsf{y}\rangle{\cdot}$$
$${\cdot}\mathcal{S}\langle \mathbf{readReq} \mid \top\rangle{\cdot}\mathcal{S}\langle \mathbf{writeRsp} \mid v = \mathsf{x}\rangle{\cdot}\mathcal{S}\langle \mathbf{readRsp} \mid v = \mathsf{y} \land st = \mathsf{true} \land v \neq \mathsf{x}\rangle$$

In contrast, the original trace $\Pi_{\mathtt{violateRYW}}$ only contains the last two events in $A'_{\mathtt{violateRYW}}$.

Trace refinement loop. Resolving the dependencies of a target event requires us to insert events that must precede it and events that must follow it; the former constitutes the *backward* synthesis phase of the algorithm, while the latter is subsumed by a *forward* synthesis pass. The two phases are symmetric: if the prophecy automaton of the message handler for the operation $\mathbf{op}_{\mathtt{parent}}$ includes the operation $\mathbf{op}_{\mathtt{child}}$, performing forward synthesis on $\mathbf{op}_{\mathtt{parent}}$ is the same as performing backward synthesis on $\mathbf{op}_{\mathtt{child}}$. To avoid repeatedly targeting the same event, our algorithm maintains two sets of events Θ_{fw} and Θ_{bw}, these sets keep track of events whose child and parent dependencies have already been resolved, resp.[8] The intersection of these two sets $(\Theta_{\mathsf{fw}} \cap \Theta_{\mathsf{bw}})$ contains those events that correspond to well-typed handlers in a controller program.

Θ_{fw} and Θ_{bw} are empty (line 1) when the refinement loop (lines 3-9 of Algorithm 1) starts, and the type context consists of ghost variables whose qualifiers are $\top$ (line 2). During each iteration, a symbolic event in the current abstract trace Π is selected ($\langle \mathbf{op} \mid \phi\rangle \notin \Theta_{\mathsf{fw}} \cap \Theta_{\mathsf{bw}}$ on line 3); the algorithm terminates once the dependencies of all symbolic events in Π are resolved. Π is partitioned into the history and future traces, Π_f and Π_f, that surround the target event. The algorithm tries to perform forward (resp., backward) synthesis on these traces, if the event is not in Θ_{fw} (resp., Θ_{bw}). If the target operation is generable (Generable(**op**) on line 7), it is the root of a dependency chain, so no additional backward synthesis is required and the event is simply added to Θ_{bw} (line 8). Both

[8]To enable Θ_{fw} and Θ_{bw} to distinguish distinct occurrences of events with the same effect operator in the abstract trace, we tag each occurence of an operator with a unique identifier. For example, $\Pi_{\mathtt{violateRYW}}$ with identifiers can be $\mathcal{S}\langle \mathbf{writeReq_1} \mid v = \mathsf{x}\rangle{\cdot}\mathcal{S}\langle \mathbf{writeReq_2} \mid v = \mathsf{y}\rangle....$

Algorithm 2: Forward Synthesis

```
1 Procedure Forward($\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f$)
      // Select the PAT of op from handler context
2     if $\Delta(\mathbf{op}) = \langle \overline{z{:}b} \dashrightarrow \overline{y{:}t} \rightarrow [H][\mathcal{S}\langle \mathbf{op} \mid \phi' \rangle][F], B \rangle$ then
3         $\Gamma \leftarrow \Gamma, \overline{z{:}\{v{:}b \mid \top\}}, \overline{y{:}t}$ ; // add ghost variables and parameters types to type context
4         $\langle \mathbf{op} \mid \phi \rangle \leftarrow \langle \mathbf{op} \mid \phi \land \phi' \rangle$ ; // merge current automata
5         $\Pi_h \leftarrow \Pi_h \land H$ ; // merge history automata
6         $\Pi_f \leftarrow \Pi_f \land F$ ; // merge prophecy automata
          // non-emptiness check
7         if $\Gamma \vdash (\Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f) \not\subseteq \neg\Box\langle \top \rangle$ then
              // return type context, property, as well as updated forward and backward set
8             return $(\Gamma, \Theta_{\mathsf{fw}} \cup \{\mathbf{op}\}, \Theta_{\mathsf{bw}} \cup B, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f)$
```

Algorithm 3: Backward Synthesis

```
1 Procedure Backward($\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f,$)
      // Choose an op_parent that sends op and retrieve its PAT from the handler context
2     if $\Delta(\mathbf{op}_{\mathsf{parent}}) = \langle \overline{z{:}b} \dashrightarrow \overline{y{:}t} \rightarrow [H][\langle \mathbf{op}_{\mathsf{parent}} \mid \phi_{\mathsf{parent}} \rangle][F_1 \cdot \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle \cdot F_2], \{\mathbf{op}\} \cup \Theta \rangle$ then
3         $\Gamma \leftarrow \Gamma, \overline{z{:}\{v{:}b \mid \top\}}, \overline{y{:}t}$ ; // add ghost variables and parameters types to the type context
4         $\langle \mathbf{op} \mid \phi \rangle \leftarrow \langle \mathbf{op} \mid \phi \land \phi' \rangle$ ; // merge current automata
5         $\Pi_h \leftarrow \Pi_h \land (H \cdot \mathcal{S}\langle \mathbf{op}_{\mathsf{parent}} \mid \phi_{\mathsf{parent}} \rangle \cdot F_1)$ ; // merge history automata
6         $\Pi_f \leftarrow \Pi_f \land F_2$ ; // merge prophecy automata
          // non-emptiness check
7         if $\Gamma \vdash (\Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f) \not\subseteq \neg\Box\langle \top \rangle$ then
              // return type context, property, as well as updated forward and backward set
8             return $(\Gamma, \Theta_{\mathsf{fw}} \cup \{\mathbf{op}_{\mathsf{parent}}\}, \Theta_{\mathsf{bw}} \cup \{\mathbf{op}\} \cup \Theta, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f)$
```

the forward and backward synthesis routines yield a 6-tuple $(\Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op}(\overline{x}) \rangle, \Pi_f)$ that contains updated history and future traces; the refined abstract trace at the end of an iteration (line 9) is simply the concatenation of the refined history trace, target event, and refined future trace.

Forward and backward synthesis. The forward synthesis subroutine is shown in Algorithm 2. It first retrieves the PAT of the target operation **op** from Δ (line 2); it also uses Δ to retrieve any children (future) dependencies events of **op**. The algorithm then merges the selected PAT into the violation property piecewise. First, the occurence of the target operation in the current abstract trace **op** is aligned with its specification in Δ (line 4). Next, the algorithm merges the history and future traces with the PAT's history and future automata (lines 5 – 6). In order to guarantee the refined abstract trace contains at least one realizable trace, the algorithm checks for non-emptiness of the violation property (line 7) by ensuring the refined automata, $\Pi_h \cdot \mathcal{S}\langle \mathbf{op}(\overline{x}) \rangle \cdot \Pi_f$, is not empty, similar to WFHAF . Finally, the algorithm returns the refined type context, property, as well as updated forward and backward sets (line 8).

The backward synthesis subroutine, shown in Algorithm 3, is similar to the forward synthesis procedure but works backward from a target event, insert preceding events into Π_h to resolve parent dependencies. The change in direction results in several differences with its forward counterpart. The procedure now selects a parent operator $\mathbf{op}_{\mathsf{parent}}$ whose handler specification has a prophecy automata that includes the target operator **op** (line 2). The refined abstract trace needs to align the target operator **op** with its counterpart in the prophecy automata of $\textbf{op}_{\textbf{parent}}$:

$$\underbrace{[H][\mathcal{S}\langle \textbf{op}_{\textbf{parent}} \mid \phi_{\text{parent}}\rangle][F_1}_{\text{actual history}} \cdot \underbrace{\mathcal{S}\langle \textbf{op} \mid \phi'\rangle}_{\text{actual current}} \cdot \underbrace{F_2]}_{\text{actual prophecy}}$$

This is reflected in how the two are merged (line 4 - 6). Finally, $\textbf{op}_{\textbf{parent}}$ and **op** are added to the forward and backward sets (line 8).

Example 4.4. We demonstrate the first step of how $A_{\texttt{violateRYW}}$ is refined into $A'_{\texttt{violateRYW}}$. The refinement loop begins in the following state:

$$\Gamma \equiv \mathsf{x}{:}\{v{:}\mathsf{int} \mid \top\}, \mathsf{y}{:}\{v{:}\mathsf{int} \mid \top\} \quad \Theta_{\mathsf{fw}} \equiv \emptyset \quad \Theta_{\mathsf{bw}} \equiv \emptyset$$

$$\Pi \equiv \Box\langle\top\rangle\cdot\mathcal{S}\langle \textbf{writeRsp} \mid v = \mathsf{x}\rangle\cdot\Box\neg\langle \textbf{writeRsp} \mid \top\rangle\cdot\mathcal{S}\langle \textbf{readRsp} \mid v = \mathsf{y} \land st = \mathsf{true} \land \mathsf{y} \neq \mathsf{x}\rangle\cdot\Box\langle\top\rangle \quad \text{(before iteration 1)}$$

The first iteration targets the **readRsp** operation. Since Θ_{fw} is empty, the algorithm performs forward synthesis on **readRsp**. No additional events are generated by the handler of **readRsp**, so no events are added to the abstract trace. Since **readRsp** is not generable, the algorithm next performs backward synthesis. The signature of **readReq** in Δ uses an intersection PAT whose branches both include **readRsp**:

$$\mathsf{x}{:}\mathsf{int} \dashrightarrow [\Diamond\langle \textbf{writeReq} \mid v = \mathsf{x}\rangle \land \neg\bigcirc\Diamond\langle \textbf{writeReq} \mid \top\rangle][\mathcal{S}\langle \textbf{readReq} \mid \top\rangle][\Diamond\langle \textbf{readRsp} \mid v = \mathsf{x} \land st = \mathsf{true}\rangle] \quad (\tau_1)$$

$$[\neg\Diamond\langle \textbf{writeRsp} \mid \top\rangle][\mathcal{S}\langle \textbf{readReq} \mid \top\rangle][\Diamond\langle \textbf{readRsp} \mid st = \mathsf{false}\rangle] \quad (\tau_2)$$

The prophecy automaton of the second branch, τ_2, requires **readRsp** to have a false status, which is at odds with the current abstract trace. This inconsistency is detected by the non-emptiness check, so we backtrack and select the next branch, τ_1. This PAT can be merged with the current trace, resulting in the following updated values of the refinement loop's variables:

$$\Gamma \equiv \mathsf{x}{:}\{v{:}\mathsf{int} \mid \top\}, \mathsf{y}{:}\{v{:}\mathsf{int} \mid \top\} \quad \Theta_{\mathsf{fw}} \equiv \{\textbf{readRsp}, \textbf{readReq}\} \quad \Theta_{\mathsf{bw}} \equiv \{\textbf{readRsp}\}$$

$$\Pi \equiv \Box\langle\top\rangle\cdot\mathcal{S}\langle \textbf{writeReq} \mid v = \mathsf{y}\rangle\cdot\Box\neg\langle \textbf{writeReq} \mid \top\rangle\cdot\mathcal{S}\langle \textbf{readReq} \mid v = \mathsf{y}\rangle\cdot\Box\langle\top\rangle\cdot\mathcal{S}\langle \textbf{writeRsp} \mid v = \mathsf{x}\rangle\cdot$$
$$\Box\neg\langle \textbf{writeRsp} \mid \top\rangle\cdot\mathcal{S}\langle \textbf{readRsp} \mid v = \mathsf{y} \land st = \mathsf{true} \land \mathsf{y} \neq \mathsf{x}\rangle\cdot\Box\langle\top\rangle \quad \text{(after iteration 1)}$$

The refined trace now includes events for **writeReq** and **readReq**, and the values of the forward and backwards sets enable both events to be targeted by the next iteration of the loop.

THEOREM 4.5 (SYNTHESIS IS SOUND). *The controller synthesized by the algorithm is type-safe with respect to our declarative typing rules.*

5 Implementation And Evaluation

We have implemented a tool based on the above approach, called Clouseau, that targets reactive distributed system models (i.e., message-passing systems defined as a collection of actors). Clouseau consists of approximately 14K lines of OCaml code and uses Z3 [7] as its backend SMT solver.[9]

Evaluation setting. Clouseau takes two inputs: a target safety property, expressed in symbolic LTL$_f$, and a handler context, Δ, that captures the behavior of actors in terms of PATs. During synthesis, Clouseau first negates the target property in order to capture unsafe traces (e.g., $A_{\texttt{violateRYW}}$), and then explores the space of possible controllers, looking for those that can guide executions towards those that are both unsafe and consistent with Δ. Each controller synthesized by Algorithm 1 fixes a particular message order for generable (i.e., environment) messages, so Clouseau systematically explores the space of alternative orderings, returning the set of all controllers found within a user-provided time bound.

We evaluate our approach by integrating our synthesized controllers into the testing framework provided by P [8, 9], a state-machine based, actor-style programming language tailored for modeling distributed systems and testing user-defined safety and liveness properties. In the P

[9]The supplemental material provides additional explanation of our experiments as well as a docker image that contains the source code of Clouseau and our benchmarks.

Table 1. Experimental results of using Clouseau to synthesize controllers for reactive distributed systems. Benchmarks from prior work are annotated with their source: P [9](†), ModP [11]($^{\diamond}$) an extension of P with support for modules, and MessageChain [27](*), an automated verification tool for P. We also include a real-world model of a two-phase commit protocol (Anon2PCModel$^{\square}$) used by a major cloud vendor. The components under test are written in P, and handler specifications are given as PATS. Clouseau can synthesize a set of controllers, each of which specifies a distinct scheduling order for messages, all consistent with provided specifications. We set a 2 minute time bound for the synthesis procedure (t_{total} is the average time to find a single controller.) We set a bound of 10K executions for the P baselines.

Benchmark	`#op`	#qualifier	#var	`#gen`	`#obs`	`#assert`	# Num. Executions			t_{total}(s)	#SMT	#fw	#bw
							Clouseau	P+Rand	P+M				
Database	4	9	6	3	3	4	1	6	-	2.73	420	4	6
Firewall*	5	21	12	2	8	9	1	12	-	5.48	788	5	8
RingLeaderElection*	3	21	12	2	6	8	1	21	-	6.53	964	2	18
EspressoMachine†	13	4	1	2	8	1	4	40	4	1.13	165	2	11
BankServer†	6	18	15	2	3	5	1	40	2	8.31	1191	2	5
Simplified2PC†	9	17	7	2	6	5	2	133	6	6.87	1043	3	8
HeartBeat†	7	18	9	4	10	9	1	61	7	7.08	1073	4	20
ChainReplication$^{\diamond}$	7	36	26	4	9	10	1	670	400	27.07	4016	6	19
Paxos$^{\diamond}$	10	32	36	4	10	13	1	Timeout	667	59.98	8763	4	16
Raft	9	32	29	3	14	14	1	Timeout	-	56.07	8356	10	22
Anon2PCModel$^{\square}$	17	73	36	4	10	10	1	Timeout	53	60.36	9023	6	12

framework, actors are executable programs that communicate via message passing. To test a system, P's runtime monitors message traffic between actors, checking that global safety and liveness properties are maintained. By default, P's runtime scheduler systematically explores arbitrary message interleavings during execution.

To test our synthesized controller with P handlers while also retaining scheduling control, we translate our controllers into a special P component that coordinates the messages between the actors under test. In this setup, all messages are routed to our controller, where they are buffered and then forwarded to the actual actors according to the order found in the synthesized output. The order in which messages are forwarded from the controller is determined by **`obs`** statements in the controller program, allowing it to control message scheduling. The coordinator is also responsible for generating and sending messages from the environment (e.g., a logical user) to the actors under test, again respecting the order in which these messages appear in the synthesized program. The assume statements in the translated coordinator ensure that generated messages always have the expected payloads; assertion failures indicate that the system under test did not encounter the potential bug in this execution, indicating the need for another attempt.

Our experimental evaluation addresses three research questions:

Q1: Is Clouseau *expressive*? Can it synthesize controllers for a diverse set of distributed protocols with realistic safety and liveness properties?

Q2: Is Clouseau *effective*? Do synthesized controller programs enable more targeted exploration of the state space to witness violations of provided safety properties than existing techniques?

Q3: Is Clouseau *efficient*? Is it able to synthesize meaningful controller programs in a reasonable amount of time?

We have evaluated Clouseau on a corpus of complex reactive system models written in P drawn from a variety of sources (described in the caption of Table 1); all of the models except for Database and Raft were written by others (**Q1**). We test the correctness of these models against a number of consistency and safety properties, including the Read-Your-Writes (RYW) consistency policy described in Sec. 2, eventual consistency, strong consistency, and unique leader invariants (as defined by RingLeader and Paxos). We introduce synthetic faults into these models manually and expose subtle bugs that can be triggered under specific message orderings or with specific input message contents. While these are synthetic design bugs, they are nonetheless representative of real and plausible errors that can be introduced when designing these models, as was illustrated in Sec. 2.

Table 1 divides the results of our experiments into four categories, separated by double bars. The first measures the complexity of benchmarks with respect to the number of distinct operators (#**op**) and the number of qualifiers used in Pat specifications and the property expressed in symbolic LTL$_f$. Our results show that we are able to specify controller-relevant behavior using anywhere from 3 - 17 different operators and 4 - 73 different kinds of qualifiers (**Q1**).

The second group of columns describes characteristics of the controllers synthesized by Clouseau, including the number of variables (#var) in the program, the number of **gen** (#**gen**) and **obs** (#**obs**) messages, and assertions (#**assert**) used in the program. Our synthesized controllers have anywhere from 1 - 36 variables, 5 - 17 messages in total, and 1 - 14 assertions. Note that the size of synthesized programs is roughly proportional to the complexity of the benchmarks (**Q2**), where the number of qualifiers correlate with the number of variables and assertions. As mentioned in Sec. 4, our algorithm is biased towards synthesizing shorter controller programs, avoiding synthesizing messages that do not directly affect the property of interest.

The third group of columns compares the performance results of our synthesized controller compared to two baselines. The first (P+Rand) uses the default P controller to generate input messages and message orderings. This baseline uses random input generation and enumerative state exploration to construct schedules, independently of the behaviors of the actors under test or the target property. The second baseline (P+M) uses manually written variants of the original model which inject additional actors into the model to control input message generation and prevent uninteresting message orderings.[10] These components play a similar role to our synthesized controllers, albeit without the benefit of rigorous specifications to help guide their definitions. The column shows the number of executions that were necessary to manifest a property violation for both baselines, as well as Clouseau. For the P baseline, we fix a bound on the number of executions to be explored to be 10K. Our results demonstrate that Clouseau consistently identifies faulty executions using only a small number of executions (fewer than 4 across all benchmarks). As benchmark complexity increases, Clouseau's effectiveness grows more apparent when compared to the default P baseline (often by many orders of magnitude). Indeed, for any of the benchmarks that only use deterministic handlers, i.e., handlers whose output messages are uniquely determined by its inputs, the synthesis procedure is always able to construct a controller that yields a property violating schedule in a single execution; for benchmarks that use internal non-determinism (e.g., EspressoMachine simulates a coffee machine that can non-deterministically fail because the machine runs out of water or beans), a small number of additional runs were required to explore different possible paths. Not surprisingly, manually crafted P environments (P+M) can improve upon the purely random baseline, but even here may sometimes require hundreds of executions to manifest a bug (e.g., the ChainReplication and Paxos benchmarks), compared to the single execution that

[10]Benchmarks from Message Chain (⋆) as well as the two benchmarks we authored (Database and Raft) do not provide these refined models.

Clouseau generates. In summary, Clouseau is able to synthesize controllers that emit executions targeted to the violation property significantly more effectively than the two baselines (**Q2**).

The last group of columns in Table 1 provides details on the cost of our synthesis procedure. The first column presents total synthesis time (t_{total}), which takes anywhere from 1.13 to 60.39 seconds (**Q3**); synthesis time is proportional to benchmark complexity, as reflected in the #**op** and #qualifier columns. The last three columns additionally analyze the behaviors of Clouseau with respect to the number of SMT queries (#SMT), as well as the number of forward synthesis (#fw) and backward synthesis steps (#bw) performed by the property refinement loop. Unsurprisingly, generating more SMT queries results in longer synthesis times; the number of these queries directly depends on the number of iterations of the property refinement loop (i.e., the sum of #fw and #bw). Oftentimes, the number of forward and backward synthesis steps exceeds the total number of messages in the controller program because Clouseau may need to backtrack when a wrong type or message interleaving is selected, which future iterations cannot resolve.

Case study. To demonstrate that Clouseau can be effective in real-world scenarios, we have applied it to Anon2PCModel, a model of a two-phase commit (2PC) protocol that is currently in use at a major cloud provider. The original P model checks a standard consistency property for 2PC transactions, specifically that if there exists a key k updated within an active transaction i, any successful read response asking its value should return the value last written to k made by that transaction. We can express a violation of this property in LTL_f as:

$$\Diamond(\langle \textbf{updateRsp} \mid tid = \mathtt{i} \land key = \mathtt{k} \land v = \mathtt{x} \land st = \mathtt{OK}\rangle \land \\ \bigcirc\neg\langle \textbf{updateRsp} \mid tid = \mathtt{i} \land key = \mathtt{k} \land st = \mathtt{OK}\rangle \,\mathcal{U}\, \langle \textbf{readRsp} \mid tid = \mathtt{i} \land key = \mathtt{k} \land v \neq \mathtt{x} \land st = \mathtt{OK}\rangle)$$

where the field *tid* represents the transaction id, while other fields have the same meanings as in the example from Sec. 1. Generating a fault-inducing scenario requires (a) initiating a new transaction with transaction id i, (b) successfully performing a write within that transaction, and then (c) subsequently performing a read within i that yields a different value than the one last written. This is an extremely challenging sequence of steps for a controller to automatically generate absent guidance from the property it is trying to violate. In contrast, since the Pat for **readReq** includes a history automaton $\Diamond\langle \textbf{startTxnRsp} \mid tid = \mathtt{i}\rangle$ that requires the user to have previously received a valid transaction id i, Clouseau can directly synthesize a controller program that strategically requests a new transaction to initiate triggering the intended violation. A version of the benchmark in which this sequence structure is enforced by a manually crafted environment can discover the violation in 53 executions, but at the cost of more user effort and a less concise model definition.

6 Related Work

Verification. Formally proving the correctness of distributed protocols and models has long been a topic of significant interest [17, 20, 40]. These approaches provide strong correctness guarantees at the cost of significant investment on the part of the proof engineer, who is responsible for, e.g., defining suitable inductive invariants for the verification task [26, 33, 45]. In contrast, our focus in this work is to improve the effectiveness of falsification techniques— validating the presence of bugs in a distributed protocol design, rather than their absence. In this sense, we are more closely related to recently proposed approaches for formally reasoning about incorrectness [25, 30, 34, 35]. While Clouseau cannot verify the correctness of a model, the burden we impose on test engineers, i.e., providing handler specifications as PATs, as well as a global safety/liveness property in LTL_f, is significantly less than what is required to verify full functional correctness of these designs.

Testing. Outside of the aforementioned P language [9, 11], several other efforts have considered how to improve the capabilities of testing frameworks for distributed systems. Jepsen [18] is a randomized testing system that seeks to reveal bugs when an application is deployed on a weakly-consistent storage system; Ozkan et al. [32] defines a randomized testing procedure for message-passing distributed systems with guaranteed lower bounds on the probability of finding a depth-d bug, where d is the minimum length of the sequence of events sufficent to witness the error. Morpheus [46] uses partial order sampling and conflict analysis to control scheduling decisions. MonkeyDB [36] uses a demonic scheduling mechanism to expose safety violations in SQL applications that interact with a weakly-isolated storage backend. Clotho [36] combines static analysis with a bounded model-checker to generate tests that expose serializability violations in weakly-consistent database systems. While these efforts are all agnostic to the property under test, Clouseau's property-guided synthesis procedure derives a controller specialized to the target property and handler specifications that capture temporal dependencies between actors. In this sense, our approach can be seen as a form of property-based testing (PBT) [4, 14] applied to open reactive systems. Broadly related to our approach is Mocket [44], a PBT-style testing framework that uses the state space graph extracted from model-checking TLA+ specifications [22] to force executions to follow specific paths in the graph. Unlike Clouseau, Mocket requires manual instrumentation of implementations to align actions defined in the specification with the corresponding code in the implementation, and relies on the TLC model-checker to produce the state space graph. In constrast, Clouseau uses a compositional refinement type system to drive synthesis, and requires no instrumentation or *a priori* enumeration of the state space to synthesize its controllers.

Specifications. TLA+ [22] is a specification language based on LTL for modeling finite-state distributed systems; the correctness of these specifications are verified using the TLC explicit-state model checker. TLA+ and its associated tooling has had notable real-world impact [29]. While Clouseau's use of LTL_f specifications in Pats is a point of commonality with TLA+, the integration of these specifications within a refinement type system, their role in driving a component-based synthesis procedure, and the top-down (TLA) vs. bottom-up (Clouseau) exploration mechanism, differentiates Clouseau's motivation and design from TLA+ and TLC in obvious ways. Type and effect systems that target *temporal* properties on the sequences of effects that a program may *produce* is a well-studied subject. For example, Skalka and Smith [41] presents a type and effect system for reasoning about the shape of histories (i.e., finite traces) of events embedded in a program. Koskinen and Terauchi [21] present a type and effect system that additionally supports verification properties of infinite traces, specified as Büchi automata. More recently, Sekiyama and Unno [39] have considered how to support richer control flow structures, e.g., delimited continuations, in such an effect system. Closest to our work are the recently proposed *Hoare Automata Types* (HATs) [47], which integrate of symbolic finite automata into a refinement type system. HATs enable reasoning about stateful sequential programs structured as a functional core interacting with opaque effectful libraries. Pats extend HATs in important ways, most notably their use of prophecy automata, which enables their use in a distributed setting in which constraints on the history of previous messages as well as requirements of future messages that have yet to be handled.

7 Conclusions

This paper proposes a property-guided testing framework for open reactive distributed system models. Our key innovation is the use of prophecy automata types (Pats) to enable the specification of message handlers in terms of history and future traces. Our component-based synthesis procedure leverages Pats to output bespoke test controllers specialized to generate executions that violate a given property. Experimental results on a wide range of benchmarks, including real-world models used in production, show that Clouseau is significantly more effective in uncovering design bugs than the existing state-of-the-art.

References

[1] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123, 2015. doi: 10.1016/J.JLAMP.2014.02.001. URL https://doi.org/10.1016/j.jlamp.2014.02.001.

[2] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, 2021.

[3] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535848.

[4] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

[5] Loris D'Antoni and Margus Veanes. Minimization of Symbolic Automata. *SIGPLAN Not.*, 49(1):541–553, Jan 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535849. URL https://doi.org/10.1145/2578855.2535849.

[6] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 854–860. AAAI Press, 2013. ISBN 9781577356332.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24.

[8] Ankush Desai and Shaz Qadeer. P: modular and safe asynchronous programming. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2017. doi: 10.1007/978-3-319-67531-2_1. URL https://doi.org/10.1007/978-3-319-67531-2_1.

[9] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM, 2013. doi: 10.1145/2491956.2462184. URL https://doi.org/10.1145/2491956.2462184.

[10] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional Programming and Testing of Dynamic Distributed Systems. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276529. URL https://doi.org/10.1145/3276529.

[11] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276529. URL https://doi.org/10.1145/3276529.

[12] Loris D'Antoni and Margus Veanes. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification*, pages 47–67. Springer, 2017.

[13] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 599–612, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009851. URL https://doi.org/10.1145/3009837.3009851.

[14] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-Based Testing in Practice. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 187:1–187:13. ACM, 2024. doi: 10.1145/3597503.3639581. URL https://doi.org/10.1145/3597503.3639581.

[15] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371080. URL https://doi.org/10.1145/3371080.

[16] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 344–358, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454048. URL https://doi.org/10.1145/3453483.3454048.

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery. doi:

10.1145/2815400.2815428. URL https://doi.org/10.1145/2815400.2815428.

[18] Jepsen. Jepsen, 2018. URL https://jepsen.io/.

[19] Ranjit Jhala and Niki Vazou. Refinement Types: A Tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021. doi: 10.1561/2500000032. URL https://doi.org/10.1561/2500000032.

[20] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973430.1973448.

[21] Eric Koskinen and Tachio Terauchi. Local temporal reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328869. doi: 10.1145/2603088.2603138. URL https://doi.org/10.1145/2603088.2603138.

[22] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X. URL http://research.microsoft.com/users/lamport/tla/book.html.

[23] Leslie Lamport. Paxos Made Simple, Fast, and Byzantine. In Alain Bui and Hacène Fouchal, editors, *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

[24] Leslie Lamport and Stephan Merz. Prophecy Made Simple. *ACM Trans. Program. Lang. Syst.*, 44(2), April 2022. ISSN 0164-0925. doi: 10.1145/3492545. URL https://doi.org/10.1145/3492545.

[25] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–27, 2022. doi: 10.1145/3527325. URL https://doi.org/10.1145/3527325.

[26] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359651. URL https://doi.org/10.1145/3341301.3359651.

[27] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi: 10.1145/3622876. URL https://doi.org/10.1145/3622876.

[28] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 759–768, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209204. URL https://doi.org/10.1145/3209108.3209204.

[29] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses Formal Methods. *Commun. ACM*, page 66–73, March 2015.

[30] Peter W. O'Hearn. Incorrectness Logic. *Proc. ACM Program. Lang.*, 4(POPL), 2019. doi: 10.1145/3371078. URL https://doi.org/10.1145/3371078.

[31] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[32] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized Testing of Distributed Systems with Probabilistic Guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[33] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016. doi: 10.1145/2908080.2908118. URL https://doi.org/10.1145/2908080.2908118.

[34] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, page 225–252, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-53290-1. doi: 10.1007/978-3-030-53291-8_14. URL https://doi.org/10.1007/978-3-030-53291-8_14.

[35] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory (CONCUR 2023)*, volume 279 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-299-0. doi: 10.4230/LIPIcs.CONCUR.2023.25. URL https://drops.dagstuhl.de/opus/volltexte/2023/19019.

[36] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. CLOTHO: directed test generation for weakly consistent database systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):117:1–117:28, 2019. doi: 10.1145/3360543. URL https://doi.org/10.1145/3360543.

[37] F. P. Ramsey. *On a Problem of Formal Logic*, pages 1–24. Birkhäuser Boston, Boston, MA, 1987. ISBN 978-0-8176-4842-8. doi: 10.1007/978-0-8176-4842-8_1. URL https://doi.org/10.1007/978-0-8176-4842-8_1.

[38] Robbert Van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. URL https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-throughput-and-availability.

[39] Taro Sekiyama and Hiroshi Unno. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571264. URL https://doi.org/10.1145/3571264.

[40] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018. doi: 10.1145/3158116. URL https://doi.org/10.1145/3158116.

[41] Christian Skalka and Scott Smith. History Effects and Verification. In Wei-Ngan Chin, editor, *Programming Languages and Systems*, pages 107–128, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30477-7.

[42] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6400-2. URL http://dl.acm.org/citation.cfm?id=645792.668302.

[43] Margus Veanes. Applications of Symbolic Finite Automata. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, pages 16–23, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39274-0.

[44] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model Checking Guided Testing for Distributed Systems. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 127–143. ACM, 2023. doi: 10.1145/3552326.3587442. URL https://doi.org/10.1145/3552326.3587442.

[45] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/yao.

[46] Xinhao Yuan and Junfeng Yang. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1141–1156, 2020.

[47] Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. A HAT Trick: Automatically Verifying Representation Invariants using Symbolic Finite Automata. *Proc. ACM Program. Lang.*, 8(PLDI):1387–1411, 2024. doi: 10.1145/3656433. URL https://doi.org/10.1145/3656433.

A Outlines of Supplemental Materials

The supplemental material is organized as follows. The complete set of rules for our operational semantics, basic typing, and declarative typing judgments are provided in Appendix B, Appendix C, and Appendix D. The type denotation is presented in Appendix E. Details of the auxiliary functions in our typing algorithm are given in Appendix F. Proofs of the theorems in our paper are provided in Appendix G. Finally, Appendix H offers a detailed explanation of our benchmarks, along with the source code of our tools and benchmarks.

B Operational Semantics

The auxiliary big-step reduction rules for effect operators and the small-step operational semantics of our core language are shown in Fig. 9.

$$\textbf{Messages} \quad m ::= \mathsf{op}(\overline{c}) \qquad \textbf{Buffers} \quad \beta \in \mathcal{P}(m) \qquad \textbf{Traces} \quad \alpha ::= [\,] \mid m :: \alpha \mid \alpha +\!\!+ \alpha$$

$$\textbf{Handler Semantics} \; \boxed{\alpha \vDash \mathsf{op}(\overline{c}) \Downarrow \beta} \qquad \textbf{Operational Semantics} \; \boxed{\phi \Downarrow c \quad \alpha \vDash (\beta, e) \xhookrightarrow{\alpha} (\beta, e)}$$

$$\frac{\beta = \{\mathsf{op}(\overline{c})\} \cup \beta_1 \quad \alpha \vDash \mathsf{op}(\overline{c}) \Downarrow \beta_2 \quad e' = e[\overline{x \mapsto c}]}{\alpha \vDash (\beta, \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e) \xhookrightarrow{[\mathsf{op}(\overline{c})]} (\beta_1 \cup \beta_2, e')}\ \textsc{StObs} \qquad \frac{\alpha \vDash \mathsf{op}(\overline{c}) \Downarrow \beta'}{\alpha \vDash (\beta, \mathbf{gen}\ \mathbf{op}\ \overline{c}\ \mathbf{in}\ e) \xhookrightarrow{[\mathsf{op}(\overline{c})]} (\beta \cup \beta', e)}\ \textsc{StGen}$$

$$\frac{op\ \overline{c} \Downarrow c_x}{\alpha \vDash (\beta, \mathbf{let}\ x = op\ \overline{c}\ \mathbf{in}\ e) \xhookrightarrow{[]} (\beta, e[x \mapsto c_x])}\ \textsc{StOp}$$

$$\frac{}{\alpha \vDash (\beta, e_1 \oplus e_2) \xhookrightarrow{[]} (\beta, e_1)}\ \textsc{StChoice1} \qquad \frac{}{\alpha \vDash (\beta, e_1 \oplus e_2) \xhookrightarrow{[]} (\beta, e_2)}\ \textsc{StChoice2}$$

$$\frac{\phi[\overline{x \mapsto c}] \Downarrow \top}{\alpha \vDash (\beta, \mathbf{assume}\ \phi\ \mathbf{in}\ e) \xhookrightarrow{[]} (\beta, e[\overline{x \mapsto c}])}\ \textsc{StAssume} \qquad \frac{\phi \Downarrow \top}{\alpha \vDash (\beta, \mathbf{assert}\ \phi\ \mathbf{in}\ e) \xhookrightarrow{[]} (\beta, e)}\ \textsc{StAssert}$$

Fig. 9. Full Operational Semantics

C Basic Typing Rules

The basic typing rules of our core language and qualifiers are shown in Fig. 10 and Fig. 11. We use an auxiliary function Ty to provide a basic type for the primitives of our language, e.g., constants, built-in operators, and data constructors.

$$\textbf{Basic Types} \quad s ::= \quad b \mid s \to s$$

Basic Typing $\boxed{\Gamma \vdash_s e : s}$

$$\frac{}{\Gamma \vdash_s c : \mathsf{Ty}(c)}\ \textsc{BtConst} \qquad \frac{\Gamma(x) = s}{\Gamma \vdash_s x : s}\ \textsc{BtVar}$$

$$\frac{\mathsf{Ty}(op) = \overline{s_i} \to s_x \quad \forall i.\Gamma \vdash_s v_i : s_i \quad \Gamma, x{:}s_x \vdash_s e : s}{\Gamma \vdash_s \mathbf{let}\ x = op\,\overline{v_i}\ \mathbf{in}\ e : s}\ \textsc{BtPureOp} \qquad \frac{\mathsf{Ty}(\mathbf{op}) = \overline{s_i} \to s_x \quad \forall i.\Gamma \vdash_s v_i : s_i \quad \Gamma \vdash_s e : s}{\Gamma \vdash_s \mathbf{gen\ op}\ \overline{v}\ \mathbf{in}\ e : s}\ \textsc{BtGen}$$

$$\frac{\mathsf{Ty}(\mathbf{op}) = \overline{s_i} \to s_x \quad \Gamma, \overline{x_i{:}s_x} \vdash_s e : s}{\Gamma \vdash_s \mathbf{let}\ \overline{x_i} = \mathbf{obs\ op\ in}\ e : s}\ \textsc{BtGen} \qquad \frac{\Gamma \vdash_s e_1 : s \quad \Gamma \vdash_s e_2 : s}{\Gamma \vdash_s e_1 \oplus e_2 : s}\ \textsc{BtChoice}$$

$$\frac{\Gamma \vdash_s \phi : \mathsf{bool} \quad \Gamma \vdash_s e : s}{\Gamma \vdash_s \mathbf{assume}\ \phi\ \mathbf{in}\ e : s}\ \textsc{BtAssume} \qquad \frac{\Gamma \vdash_s \phi : \mathsf{bool} \quad \Gamma \vdash_s e : s}{\Gamma \vdash_s \mathbf{assert}\ \phi\ \mathbf{in}\ e : s}\ \textsc{BtAssert}$$

Fig. 10. Basic Typing Rules

Basic Qualifier Typing $\boxed{\Gamma \vdash_s \phi : s}$

$$\frac{\mathsf{Ty}(c) = s}{\Gamma \vdash_s c : s}\ \textsc{BtLitConst} \qquad \frac{\Gamma(x) = s}{\Gamma \vdash_s x : s}\ \textsc{BtLitVar} \qquad \frac{}{\Gamma \vdash_s \top : \mathsf{bool}}\ \textsc{BtTop} \qquad \frac{}{\Gamma \vdash_s \bot : \mathsf{bool}}\ \textsc{BtBot}$$

$$\frac{\mathsf{Ty}(op) = \overline{s_i} \to s \quad \forall i.\Gamma \vdash_s l_i : s_i}{\Gamma \vdash_s op\,\overline{l_i} : s}\ \textsc{BtLitOp} \qquad \frac{\Gamma \vdash_s \phi : \mathsf{bool}}{\Gamma \vdash_s \neg\phi : \mathsf{bool}}\ \textsc{BtNeg}$$

$$\frac{\Gamma \vdash_s \phi_1 : \mathsf{bool} \quad \Gamma \vdash_s \phi_2 : \mathsf{bool}}{\Gamma \vdash_s \phi_1 \land \phi_2 : \mathsf{bool}}\ \textsc{BtAnd} \qquad \frac{\Gamma \vdash_s \phi_1 : \mathsf{bool} \quad \Gamma \vdash_s \phi_2 : \mathsf{bool}}{\Gamma \vdash_s \phi_1 \lor \phi_2 : \mathsf{bool}}\ \textsc{BtOr} \qquad \frac{\Gamma, x{:}b \vdash_s \phi : \mathsf{bool}}{\Gamma \vdash_s \forall x{:}b.\phi : \mathsf{bool}}\ \textsc{BtForall}$$

Fig. 11. Basic Qualifier Typing Rules

Type Erasure $\boxed{\lfloor t \rfloor \quad \lfloor \tau \rfloor \quad \lfloor \Gamma \rfloor}$

$$\lfloor \{v{:}b \mid \phi\} \rfloor \doteq b \qquad \lfloor x{:}t \rightarrow \tau \rfloor \doteq \lfloor t \rfloor \rightarrow \lfloor \tau \rfloor \qquad \lfloor x{:}b \dashrightarrow t \rfloor \doteq \lfloor t \rfloor$$

$$\lfloor [H][A][F] \rfloor \doteq \texttt{unit} \qquad \lfloor [H][A][F]\{B\} \rfloor \doteq \texttt{unit} \qquad \lfloor \tau_1 \sqcap \tau_2 \rfloor \doteq \lfloor \tau_1 \rfloor$$

$$\lfloor \emptyset \rfloor \doteq \emptyset \qquad \lfloor x{:}t, \Gamma \rfloor \doteq x{:}\lfloor t \rfloor, \lfloor \Gamma \rfloor$$

Well-formedness $\boxed{\Gamma \vdash^{\mathbf{WF}} A \quad \Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma \vdash^{\mathbf{WF}} t}$

$$\frac{\mathsf{Ty}(\mathsf{op}) = \overline{x_i{:}b_i} \rightarrow \texttt{unit} \qquad \lfloor \Gamma \rfloor, \overline{x_i{:}b_i} \vdash_{\mathsf{s}} \phi : \texttt{bool}}{\Gamma \vdash^{\mathbf{WF}} \langle \mathbf{op}\ \overline{x_i} \mid \phi \rangle}\ \textsc{WfEvent} \qquad \frac{\lfloor \Gamma \rfloor \vdash_{\mathsf{s}} \phi : \texttt{bool}}{\Gamma \vdash^{\mathbf{WF}} \langle \phi \rangle}\ \textsc{WfTest}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} A}{\Gamma \vdash^{\mathbf{WF}} \neg A}\ \textsc{WfNeg} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} A_1 \quad \Gamma \vdash^{\mathbf{WF}} A_2}{\Gamma \vdash^{\mathbf{WF}} A_1 \land A_2}\ \textsc{WfAnd} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} A_1 \quad \Gamma \vdash^{\mathbf{WF}} A_2}{\Gamma \vdash^{\mathbf{WF}} A_1 \lor A_2}\ \textsc{WfOr}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} A_1 \quad \Gamma \vdash^{\mathbf{WF}} A_2}{\Gamma \vdash^{\mathbf{WF}} A_1 ; A_2}\ \textsc{WfConcat} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} A}{\Gamma \vdash^{\mathbf{WF}} \bigcirc A}\ \textsc{WfNext} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} A_1 \quad \Gamma \vdash^{\mathbf{WF}} A_2}{\Gamma \vdash^{\mathbf{WF}} A_1\, \mathcal{U}\, A_2}\ \textsc{WfUntil}$$

$$\frac{\lfloor \Gamma \rfloor, v{:}b \vdash_{\mathsf{s}} \phi : \texttt{bool}}{\Gamma \vdash^{\mathbf{WF}} \{v{:}b \mid \phi\}}\ \textsc{WfPBase} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} t_x \qquad \Gamma, x{:}\lfloor t_x \rfloor \vdash^{\mathbf{WF}} t}{\Gamma \vdash^{\mathbf{WF}} x{:}t_x \rightarrow t}\ \textsc{WfPArr}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} H \quad \Gamma \vdash^{\mathbf{WF}} A \quad \Gamma \vdash^{\mathbf{WF}} F \\ \Gamma \vdash H \cdot A \cdot F \not\subseteq \neg\Box\langle\top\rangle\end{array}}{\Gamma \vdash^{\mathbf{WF}} [H][A][F]}\ \textsc{WfHAF}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} \tau \qquad \Gamma, x{:}\lfloor t_x \rfloor \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash^{\mathbf{WF}} x{:}t_x \rightarrow \tau}\ \textsc{WfArr} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \tau \qquad \Gamma, x{:}b \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash^{\mathbf{WF}} x{:}b \dashrightarrow \tau}\ \textsc{WfGArr} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} \tau_1 \quad \Gamma \vdash^{\mathbf{WF}} \tau_2 \\ \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor\end{array}}{\Gamma \vdash^{\mathbf{WF}} \tau_1 \sqcap \tau_2}\ \textsc{WfInter}$$

Fig. 12. Full set of well-formedness typing rules.

D Declarative Typing Rules

The full set of rules for our auxiliary typing relations are shown in Fig. 12 and Fig. 13. The full set of declarative typing rules are shown in Fig. 14. We elide the basic typing relation ($\emptyset \vdash_{\mathsf{s}} e : s$) in the premises of the rules in Fig. 14; all of these rules assume any terms they reference have a basic type.

Automata Inclusion $\boxed{\Gamma \vdash A \subseteq A}$ **Subtyping** $\boxed{\Gamma \vdash t <: t \quad \Gamma \vdash \tau <: \tau}$

$$\frac{\forall \sigma \in [\![\Gamma]\!].[\![\sigma(A_1)]\!] \subseteq [\![\sigma(A_2)]\!]}{\Gamma \vdash A_1 \subseteq A_2}\ \textsc{SubAutomata} \qquad \frac{\Gamma \vdash H_2 \subseteq H_1 \quad \Gamma \vdash A_1 \subseteq A_2 \quad \Gamma \vdash F_2 \subseteq F_1}{\Gamma \vdash [H_1][A_1][F_1] <: [H_2][A_2][F_2]}\ \textsc{SubHAF}$$

$$\frac{\Gamma \vdash H_2 \subseteq H_1 \quad \Gamma \vdash A_1 \subseteq A_2 \quad \Gamma \vdash F_2 \subseteq F_1}{\Gamma \vdash [H_1][A_1][F_1]\{B\} <: [H_2][A_2][F_2]\{B\}}\ \textsc{SubHAFB}$$

$$\frac{}{\Gamma \vdash \tau_1 \sqcap \tau_2 <: \tau_1}\ \textsc{SubIntLL} \qquad \frac{}{\Gamma \vdash \tau_1 \sqcap \tau_2 <: \tau_2}\ \textsc{SubIntLR} \qquad \frac{\Gamma \vdash \tau <: \tau_1 \quad \Gamma \vdash \tau <: \tau_2}{\Gamma \vdash \tau <: \tau_1 \sqcap \tau_2}\ \textsc{SubIntR}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_2 <: t_1 \\ \Gamma, x{:}t_2 \vdash \tau_1 <: \tau_2\end{array}}{\Gamma \vdash x{:}t_1 \rightarrow \tau_1 <: x{:}t_2 \rightarrow \tau_2}\ \textsc{SubArr} \qquad \frac{\Gamma, x{:}\{\nu{:}b \mid \top\} \vdash t_1 <: t_2}{\Gamma \vdash t_1 <: x{:}b \dashrightarrow t_2}\ \textsc{SubGhostR} \qquad \frac{\begin{array}{c}\exists v.\lfloor\Gamma\rfloor \vdash_s v : b \\ \Gamma \vdash t_1[x \mapsto v] <: t_2\end{array}}{\Gamma \vdash x{:}b \dashrightarrow t_1 <: t_2}\ \textsc{SubGhostL}$$

$$\frac{\forall \sigma.\sigma \in [\![\Gamma]\!].\sigma(\phi_1 \implies \phi_2)}{\Gamma \vdash \{\nu{:}b \mid \phi_1\} <: \{\nu{:}b \mid \phi_2\}}\ \textsc{SubPBase} \qquad \frac{\Gamma \vdash t_{x_2} <: t_{x_1} \quad \Gamma, x{:}t_{x_2} \vdash t_1 <: t_2}{\Gamma \vdash x{:}t_{x_1} \rightarrow \tau_1 <: x{:}t_{x_2} \rightarrow \tau_2}\ \textsc{SubPArr}$$

Fig. 13. Full set of subtyping rules.

Typing $\boxed{\Gamma \vdash v : t \quad \Gamma; \Delta; \Theta \vdash e : \tau}$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Theta \vdash e_1 : \tau \\ \Gamma;\Delta;\Theta \vdash e_2 : \tau\end{array}}{\Gamma;\Delta;\Theta \vdash e_1 \oplus e_2 : \tau}\ \textsc{TChoice} \qquad \frac{\begin{array}{c}\Delta(\mathbf{op}) = \langle \mathbf{gen}\ \tau, \Theta'\rangle \quad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \rightarrow [H][\mathcal{S}\langle \mathbf{op} \mid \phi\rangle][A \cdot F] \\ \forall i.\Gamma \vdash v_i : t_i \quad \Gamma;\Delta;\Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle][A][F]\end{array}}{\Gamma;\Delta;\Theta \vdash \mathbf{gen}\ \mathbf{op}\ \overline{v_i}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle \cdot A][F]}\ \textsc{TGen}$$

$$\frac{}{\Gamma;\Delta;\emptyset \vdash () : [H][\Box\langle\bot\rangle][F]}\ \textsc{TRet} \qquad \frac{\begin{array}{c}\Delta(\mathbf{op}) = \langle \mathbf{obs}\ \tau, \Theta'\rangle \quad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \rightarrow [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi\rangle][A \cdot F] \\ \Gamma, \overline{x{:}t};\Delta;\Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \wedge \overline{y = x}\rangle][A][F]\end{array}}{\Gamma;\Delta;\{\mathbf{op}\} \cup \Theta \vdash \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi\rangle \cdot A][F]}\ \textsc{TObs}$$

$$\frac{\Gamma, z{:}\{\nu{:}\mathsf{unit} \mid \phi\};\Delta;\Theta \vdash e : \tau \quad z \text{ is fresh}}{\Gamma;\Delta;\Theta \vdash \mathbf{assume}\ \phi\ \mathbf{in}\ e : \tau}\ \textsc{TAssume} \qquad \frac{\Gamma;\Delta;\Theta \vdash e : \tau \quad \Gamma \vdash () : \{\nu{:}\mathsf{unit} \mid \phi\}}{\Gamma;\Delta;\Theta \vdash \mathbf{assert}\ \phi\ \mathbf{in}\ e : \tau}\ \textsc{TAssert}$$

$$\frac{\begin{array}{c}\Gamma \vdash op : t \quad \Gamma \vdash t <: \overline{y{:}t} \rightarrow t_x\ \forall i.\Gamma \vdash v_i : t_i \\ \Gamma, x{:}t_x[\overline{y \mapsto v}];\Delta;\Theta \vdash e : \tau\end{array}}{\Gamma;\Delta;\Theta \vdash \mathbf{let}\ x{:}b = op\ \overline{v}\ \mathbf{in}\ e : \tau}\ \textsc{TOpApp} \qquad \frac{\lfloor\Gamma\rfloor \vdash_s v : b}{\Gamma \vdash v : \{\nu{:}b \mid \nu = v\}}\ \textsc{TVal}$$

$$\frac{\Gamma;\Delta;\Theta \vdash e : \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma;\Delta;\Theta \vdash e : \tau'}\ \textsc{TSub} \qquad \frac{\Gamma \vdash v : t \quad \Gamma \vdash t <: t'}{\Gamma \vdash v : t'}\ \textsc{TPureSub}$$

Fig. 14. Full set of typing rules.

E Type Denotation

Well-Formed Message, Buffer, and Trace $\boxed{\vdash^{\textbf{WF}} m \quad \vdash^{\textbf{WF}} \beta \quad \vdash^{\textbf{WF}} \alpha}$

$$\frac{\emptyset \vdash_{\mathsf{S}} \textbf{op} : \overline{b} \to \texttt{unit} \quad \forall i. \emptyset \vdash_{\mathsf{S}} c_i : b_i}{\vdash^{\textbf{WF}} \textbf{op}(\overline{c})}\ \textsc{WfMsg} \qquad \frac{\forall m \in \beta. \vdash^{\textbf{WF}} m}{\vdash^{\textbf{WF}} \beta}\ \textsc{WfBuffer}$$

$$\frac{}{\vdash^{\textbf{WF}} []}\ \textsc{WfNil} \qquad \frac{\vdash^{\textbf{WF}} m \quad \vdash^{\textbf{WF}} \alpha}{\vdash^{\textbf{WF}} m :: \alpha}\ \textsc{WfCons}$$

Trace Language $\boxed{\alpha, i \models A \quad [\![A]\!] \in \mathcal{P}(\alpha)}$

$$[\![A]\!] \doteq \{\alpha \mid \vdash^{\textbf{WF}} \alpha \land \alpha, 0 \models A\}$$

$$\alpha, i \models \langle \textbf{op}\ \overline{x} \mid \phi \rangle \iff \alpha[i] = \textbf{op}(\overline{c}) \land \phi[\overline{x \mapsto c}] \qquad \alpha, i \models A \land A' \iff \alpha, i \models A \land \alpha, i \models A'$$

$$\alpha, i \models \langle \phi \rangle \iff \alpha[i] = \textbf{op}(\overline{c}) \land \phi \qquad \alpha, i \models A \lor A' \iff \alpha, i \models A \lor \alpha, i \models A'$$

$$\alpha, i \models \bigcirc A \iff \alpha, i{+}1 \models A \qquad \alpha, i \models A_1; A_2 \iff \alpha[i...len(\alpha)] = \alpha_1 {+\!\!+} \alpha_2 \land \alpha_1 \in [\![A_1]\!] \land \alpha_2 \in [\![A_2]\!]$$

$$\alpha, i \models \neg A \iff \alpha, i \not\models A \qquad \alpha, i \models A\, \mathcal{U}\, A' \iff \exists j. i \le j < len(\alpha). \alpha, j \models A' \land \forall k. i \le k < j \implies \alpha, k \models A$$

Type Denotation $\boxed{[\![t]\!] \in \mathcal{P}(c) \quad [\![\tau]\!] \in \mathcal{P}(e)}$

$$
\begin{aligned}
[\![\{v{:}b \mid \phi\}]\!] &\doteq \{c \mid \emptyset \vdash_{\mathsf{S}} c : b \land \phi[v \mapsto v]\} \\
[\![x{:}t_x \to t]\!] &\doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor x{:}t_x \to t \rfloor \land \forall c \in [\![t_x]\!].\, e\ c \in [\![\tau[t \mapsto c]]\!]\} \\
[\![x{:}t \to \tau]\!] &\doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor x{:}t \to \tau \rfloor \land \forall c \in [\![t]\!].\, e\ c \in [\![\tau[x \mapsto c]]\!]\} \\
[\![x{:}t \dashrightarrow \tau]\!] &\doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor \tau \rfloor \land \forall c \in [\![t]\!]. e \in [\![\tau[x \mapsto c]]\!]\} \\
[\![[H][A][F]]\!] &\doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \texttt{unit} \land \forall \alpha_h \in [\![H]\!]. \forall \alpha_f \in [\![F]\!]. \forall \alpha\ \beta\ \beta'\ e_h\ e_f. \\
&\qquad [] \vDash (\emptyset, e_h) \overset{\alpha_h}{\hookrightarrow}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \alpha_h {+\!\!+} \alpha \vDash (\beta', e_f) \overset{\alpha_f}{\hookrightarrow}{}^{*} (\emptyset, ()) \implies \alpha \in [\![A]\!]\} \\
[\![\tau_1 \sqcap \tau_2]\!] &\doteq [\![\tau_1]\!] \cap [\![\tau_2]\!]
\end{aligned}
$$

Type Context Denotation $\boxed{[\![\Gamma]\!] \in \mathcal{P}(\sigma)}$

$$[\![\emptyset]\!] \doteq \{\emptyset\} \qquad [\![x{:}t, \Gamma]\!] \doteq \{\sigma[x \mapsto c] \mid c \in [\![t]\!], \sigma \in [\![\Gamma[x \mapsto c]]\!]\}$$

Capability Context Denotation $\boxed{[\![\Theta]\!] \in \mathcal{P}(\beta)}$

$$[\![\Theta]\!] \doteq \{\{\overline{\textbf{op}(\overline{c})}\} \mid \{\textbf{op}\} = \Theta \land \vdash^{\textbf{WF}} \{\overline{\textbf{op}(\overline{c})}\}\}$$

Fig. 15. Type denotations in $\boldsymbol{\lambda}^U$

F Auxiliary Functions for Synthesis

This section describes two auxiliary functions used for controller synthesis. The first of these, **Norm**, converts a a symbolic LTL$_f$ formula into a set of unsafe abstract traces, which are then given to Algorithm 1 as input. The second, **TermDerive**, generates a controller program from a refined abstract trace.

Normalization. The function **Norm** first convert an input automata expressed in symbolic LTL$_f$ into standard Negation Normal Form (NNF) on line 1, then recursively translates the input automata into a set of abstract traces. Note that the negation operator $\neg$ only appears before the atomic predicates (i.e., $\langle \textbf{op} \mid \phi \rangle$ and $\langle \phi \rangle$).

Lemma F.1. *[Abstract traces are closed under conjunction] The conjunction ($\land$) of two abstract traces is also an abstract trace.*

Algorithm 4: Abstract Trace Normalization

```
1  Procedure Norm($A$) :=
2      $A \leftarrow \texttt{ToNNF}(A)$;
3      match $A$:
4          case $\langle \mathsf{op} \mid \phi\rangle$ do return $\{\mathcal{S}\langle \mathsf{op} \mid \phi\rangle\cdot\Box\langle\top\rangle\}$ ;
5          case $\neg\langle \mathsf{op} \mid \phi\rangle$ do return $\{\mathcal{S}\langle \mathsf{op} \mid \neg\phi\rangle\cdot\Box\langle\top\rangle\} \cup \{\mathcal{S}\langle \mathsf{op}' \mid \top\rangle\cdot\Box\langle\top\rangle \mid \mathsf{op}' \neq \mathsf{op}\}$ ;
6          case $\langle\phi\rangle$ do return $\{\mathcal{S}\langle \mathsf{op} \mid \phi\rangle \mid$ for all $\mathsf{op}\}$;
7          case $\neg\langle\phi\rangle$ do return $\{\mathcal{S}\langle \mathsf{op} \mid \neg\phi\rangle \mid$ for all $\mathsf{op}\}$;
8          case $\mathcal{S}A$ do return $\{\mathcal{S}\langle \mathsf{op} \mid \phi\rangle \mid \mathcal{S}\langle \mathsf{op} \mid \phi\rangle\cdot\Pi \in \mathbf{Norm}(A)\}$ ;
9          case $\bigcirc A$ do return $\{\langle \mathsf{op} \mid \top\rangle\cdot\Pi \mid$ for all $\mathsf{op}, \Pi \in \mathbf{Norm}(A)\}$ ;
10         case $A_1 \,\mathcal{U}\, A_2$ do return $\{(\Box\neg A_1)\cdot\Pi_2 \mid \Pi_2 \in \mathbf{Norm}(A_2)\}$;
11         case $A_1\cdot A_2$ do return $\{\Pi_1\cdot\Pi_2 \mid \Pi_1 \in \mathbf{Norm}(A_1) \land \Pi_2 \in \mathbf{Norm}(A_2)\}$;
12         case $\Diamond A$ do return $\{\Box\langle\top\rangle\cdot\Pi\cdot\Box\langle\top\rangle \mid \Pi \in \mathbf{Norm}(A)\}$;
13         case $\Box A$ do return $\{\Box A\}$;
14         case $A_1 \lor A_2$ do return $\mathbf{Norm}(A_1) \cup \mathbf{Norm}(A_2)$;
15         case $A_1 \land A_2$ do return $\{\Pi_1 \land \Pi_2 \mid \Pi_1 \in \mathbf{Norm}(A_1) \land \Pi_2 \in \mathbf{Norm}(A_2)\}$;
```

Algorithm 5: Term Derivation

```
1  Procedure TermDerive($\Gamma, \Pi$) :=
2      match $\Gamma$:
3          case [] do
4              return DeriveTrace($\Pi$);
5          case $x{:}\{v{:}b \mid \phi\} :: \Gamma'$ do
6              return assume $\phi[v \mapsto x]$ in TermDerive($\Gamma'$, ToList($\Pi$));
```

Algorithm 6: Trace Derivation

```
1  Procedure DeriveTrace($\Pi$) :=
2      match $\Pi$:
3          case [] do return () ;
4          case $\Box A :: \Pi'$ do return DeriveTrace($\Pi'$) ;
5          case $\mathcal{S}\langle \mathsf{op}\ \overline{x} \mid \phi\rangle\cdot\Pi'$ when gen op do
6              $\overline{x'} \leftarrow \texttt{GetFreshNames}(\overline{x})$;
7              assume $\phi[\overline{x} \mapsto \overline{x'}]$ in gen op $\overline{x'}$ in DeriveTrace($\Pi'$);
8          case $\mathcal{S}\langle \mathsf{op}\ \overline{x} \mid \phi\rangle :: \Pi'$ when obs op do
9              $\overline{x'} \leftarrow \texttt{GetFreshNames}(\overline{x})$;
10             let $\overline{x'}$ = obs op in assert $\phi[\overline{x \mapsto x'}]$ in DeriveTrace($\Pi'$);
```

Lemma F.2. *[Normalization is sound] The normalized result has the same denotation as the input automata, that is, for all automata A and set of traces $\{\Pi_i\}$,*

$$[\![A]\!] = \bigcup_i [\![\Pi_i]\!]$$

Term Derivation. The term derivation function **TermDerive** is shown in Algorithm 5. It first converts the input type context into **`assume`** statements over the corresponding qualifiers in pure refinement types (line 6), then derives the abstract trace with the help of the **DeriveTrace** subroutine shown in Algorithm 6. The input abstract trace is first be converted into a list of automata (`ToList`) before the subroutine is called; it then recursively transforms this list into a controller program. Note that our algorithm prioritizes shorter controller programs, so **DeriveTrace** skips automata with global modality ($\square A$) on line 4. For a generable symbolic event (line 5), **DeriveTrace** inserts an **`assume`** expression before the **`gen`** expression on line 7. Conversely, for observable events, **DeriveTrace** adds an **`assert`** expression after the **`obs`** expression on line 10.

Lemma F.3. *[Term Derivation is Sound] Fora given type context* Γ, *well-founded type context* Δ, *abstract trace* Π, *and term* e,

$$(\exists e'.\Gamma;\Delta;\emptyset \vdash e' : [\square\langle\bot\rangle][A][\square\langle\bot\rangle]) \implies \textbf{TermDerive}(\Gamma,\Pi) = e \implies \Gamma;\Delta;\emptyset \vdash e : [\square\langle\bot\rangle][A][\square\langle\bot\rangle]$$

G Proofs

We omit the completely standard proof that basic typing $\vdash_s e : s$ is sound, assuming that all terms and qualifiers in our typing rules and theorems are type-safe. Before presenting the proof of the fundamental theorem and type soundness, we introduce several useful lemmas.

G.1 Lemmas

G.1.1 Common symbolic LTL_f formulas.

Lemma G.1. . *$\Box\langle\top\rangle$ contains all well-formed traces.* $[\![\Box\langle\top\rangle]\!] = \{tr \mid \vdash^{\textbf{\textit{WF}}} tr\}$.

Lemma G.2. . *$\Box\langle\bot\rangle$ only contains the empty trace.* $[\![\Box\langle\bot\rangle]\!] = \{[\,]\}$.

Lemma G.3. *$\neg\Box\langle\top\rangle$ contains no traces.* $[\![\neg\Box\langle\top\rangle]\!] = \emptyset$.

G.1.2 Denotations.

Lemma G.4. *[Denotation of singleton modality] For all symbolic event $\langle \textbf{op}\ \overline{x_i} \mid \phi\rangle$ and values $\overline{v_i}$,*

$$\phi[\overline{x_i \mapsto v_i}] \implies [\textbf{op}(\overline{v_i})] \in [\![\mathcal{S}\langle \textbf{op}\ \overline{x_i} \mid \phi\rangle]\!]$$

Lemma G.5. *[Denotation of concatenation] For all automata A_1 and A_2 and trace α,*

$$\alpha \in [\![A_1 \cdot A_2]\!] \iff (\exists \alpha_1\ \alpha_2. \alpha = \alpha_1 +\!\!+ \alpha_2 \land \alpha_1 \in [\![A_1]\!] \land \alpha_2 \in [\![A_2]\!])$$

Lemma G.6. *[Denotation of choice] For all term e_1 and e_2 and* Pat *τ,*

$$e_1 \in [\![\tau]\!] \land e_2 \in [\![\tau]\!] \implies e_1 \oplus e_2 \in [\![\tau]\!]$$

Lemma G.7. *[Denotation of pure computation] For all term e_1 and e_2 and* Pat *τ,*

$$(\forall \alpha\ \beta. \alpha \vDash (\beta, e) \overset{[\,]}{\hookrightarrow}{}^{*} (\beta, e')) \implies e \in [\![\tau]\!] \iff e' \in [\![\tau]\!]$$

Lemma G.8 (Buffer Partition). *For all capability Θ, automata F and buffer β, we have*

$$\beta \in [\![\Theta_1 \cup \Theta_2]\!] \iff \exists \beta_1\ \beta_2. \beta_1 \cup \beta_2 = \beta \land \beta_1 \cap \beta_2 = \emptyset \land \beta_1 \in \Theta_1 \land \beta_2 \in \Theta_2$$

G.1.3 Subtyping.

Lemma G.9. *[Pure Subtyping Soundness] For Given type context Γ and well-formed pure refinement type t and t': $\Gamma \vdash t <: t' \implies \forall \sigma \in [\![\Gamma]\!].[\![\sigma(t)]\!] \subseteq [\![\sigma(t')]\!]$*

Lemma G.10. *[Subtyping Soundness] For Given type context Γ and well-formed* Pat *τ and τ': $\Gamma \vdash \tau <: \tau' \implies \forall \sigma \in [\![\Gamma]\!].[\![\sigma(\tau)]\!] \subseteq [\![\sigma(\tau')]\!]$*

G.1.4 Substitution.

Lemma G.11 (Substitution Lemma). *For Given type context Γ, variable x, well-formed pure refinement type t,*Pat *τ and term e: $\Gamma, x{:}t; \Delta; \Theta \vdash e : \tau \implies \forall v. \Gamma \vdash v : t \implies \Gamma; \Delta; \Theta \vdash e[x \mapsto v] : \tau[x \mapsto v]$*

G.1.5 Handler Contexts.

Definition G.12 (Well-formed handler context). The handler specification Δ is well-formed iff for all operator $\textbf{op}$ and its Pat $\overline{y{:}b} \dashrightarrow \overline{x{:}t} \rightarrow [H][\mathcal{S}\langle \textbf{op}\ \overline{y} \mid \phi\rangle][F]$ and capability $\{\overline{\textbf{op}_\textbf{i}}\}$ in Δ satisfying

$$\begin{aligned}&\forall \overline{y{:}b}. \forall \alpha_h \in [\![H]\!]. \forall \overline{c \in [\![t]\!]}. \forall \overline{c_{ij}}. \forall \overline{\alpha_i}. \alpha_1 +\!\!+ [\textbf{op}_1(\overline{c_{1j}})] +\!\!+ ... [\textbf{op}_\textbf{n}(\overline{c_{nj}})] +\!\!+ \alpha_{n+1} \in [\![F]\!] \implies \\ &\quad \alpha_h \vDash \textbf{op}(\overline{c}) \Downarrow \{\textbf{op}_\textbf{i}(\overline{c_{ij}})\} \land \phi[\overline{x \mapsto c}]\end{aligned}$$

Lemma G.13 (Well-formed handler context with subsumption). *For given well-formed handler specification Δ, type context Γ, and effect operator* **op**

$$\Delta(\textbf{op}) = \langle \tau, \Theta \rangle \implies \Gamma \vdash \tau <: \overline{x{:}t} \to [H][\mathcal{S}\langle \textbf{op}\ \overline{y} \mid \phi \rangle][F] \implies$$
$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\overline{\forall c \in [\![\sigma(t)]\!]}.\forall \alpha_f \in [\![\sigma(F)]\!].\exists \overline{\alpha_i}.\exists \overline{m_i}.$$
$$\alpha_1 \mathbin{+\!\!+} [m_1] \mathbin{+\!\!+} \ldots [m_n] \mathbin{+\!\!+} \alpha_{n+1} = \alpha_f \land (\forall \textbf{op}_\textbf{i}.\textbf{op}_\textbf{i} \in \Theta \iff \exists \overline{c_i}.m_i = \textbf{op}_\textbf{i}(\overline{c_i})) \implies \alpha_h \vDash \textbf{op}(\overline{c}) \Downarrow \{m_i\} \land \sigma(\phi)[\overline{x \mapsto c}]$$

Lemma G.14 (Well-formed pure context with subsumption). *For given specification Δ, type context Γ, and pure operator op*

$$\Delta(op) = t \land \Gamma \vdash t <: \overline{y{:}t_y} \to t_x \implies \forall \sigma \in [\![\Gamma]\!].\overline{\forall c_y \in [\![\sigma(t_y)]\!]}.op(\overline{c_y}) \Downarrow c \implies c \in [\![\sigma(t_x[\overline{y \mapsto c_y}])]\!]$$

G.2 Fundamental Theorem

We first prove the fundamental theorem for values.

Theorem G.15. *[Pure Fundamental Theorem] For Given type context Γ and well-formed value v as well as pure refinement type t*: $\Gamma \vdash v : t \implies \forall \sigma \in [\![\Gamma]\!].\sigma(v) \in [\![\sigma(t)]\!]$

Proof. We proceed by induction over our type judgment $\Gamma; \Delta; \Theta \vdash e : \tau$, which has two cases proved as following:

$$\text{Case}\ :\ \frac{\lfloor \Gamma \rfloor \vdash_{\mathbf{S}} v : b}{\Gamma \vdash v : \{\nu{:}b \mid \nu = v\}}\ \textsc{TVal}$$

where we need to prove $\forall \sigma \in [\![\Gamma]\!].\sigma(v) \in [\![\{\nu{:}b \mid \nu = \sigma(v)\}]\!]$, which can be directly proved by definition of type denotation.

$$\text{Case}\ :\ \frac{\Gamma \vdash v : t \qquad \Gamma \vdash t <: t'}{\Gamma \vdash v : t'}\ \textsc{TPureSub}$$

where we have inductive hypothesis $\forall \sigma \in [\![\Gamma]\!].\sigma(v) \in [\![\sigma(t)]\!]$ and need to prove $\forall \sigma \in [\![\Gamma]\!].\sigma(v) \in [\![\sigma(t')]\!]$, which can be directly proved by soundness lemma of pure subtyping (lemma G.9).

□

The fundamental theorem for a controller program consists of two parts: (1) the history, current, and future traces of a well-typed term e are consistent with the corresponding Pat; (2) the realizability guarantee provided by the capability. We first prove the first part, as follows.

Theorem G.16. *[Fundamental Theorem For Trace Consistency] Given a well-formed handler specification Δ, the trace of effects produced by a well-typed term e is captured by its corresponding* Pat *τ*: $\Gamma; \Delta; \Theta \vdash e : \tau \implies \forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(e) \in [\![\sigma(\tau)]\!]$.

Proof. We proceed by induction over our type judgment $\Gamma; \Delta; \Theta \vdash e : \tau$, which has 8 cases proved as following:

$$\text{Case}\ :\ \frac{\begin{array}{c}\Delta(\textbf{op}) = \langle \textbf{gen}\ \tau, \Theta' \rangle \qquad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \textbf{op} \mid \phi \rangle][A{\cdot}F] \\ \forall i.\Gamma \vdash v_i : t_i \qquad \Gamma; \Delta; \Theta \cup \Theta' \vdash e : [H{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle][A][F]\end{array}}{\Gamma; \Delta; \Theta \vdash \textbf{gen op}\ \overline{v_i}\ \textbf{in}\ e : [H][\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle{\cdot}A][F]}\ \textsc{TGen}$$

This rule assume that $e \equiv \textbf{gen op}\ \overline{v}\ \textbf{in}\ e, \tau \equiv [H][\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x \mapsto v}]\rangle{\cdot}A][F]$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(\textbf{gen op}\ \overline{v}\ \textbf{in}\ e) \in [\![\sigma([H][\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x \mapsto v}]\rangle{\cdot}A][F])]\!]$$

From the induction hypothesis and the precondition of this rule, we have

$$\Delta(\mathbf{op}) = \langle \mathbf{gen}\ \tau, \Theta' \rangle \qquad \text{(assumption)} \tag{1}$$

$$\Gamma \vdash \tau <: \overline{x{:}t} \to [H][\mathcal{S}\langle \mathbf{op} \mid \phi \rangle][A\cdot F] \qquad \text{(assumption)} \tag{2}$$

$$\forall i.\Gamma \vdash v_i : t_i \qquad \text{(assumption)} \tag{3}$$

$$\Gamma \mid \Theta \cup \Theta' \vdash e : [H\cdot\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle][A][F] \qquad \text{(assumption)} \tag{4}$$

$$\forall \sigma \in [\![\Gamma]\!].\sigma(e) \in [\![\sigma([H\cdot\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle][A][F])]\!] \qquad \text{(induction hypothesis)} \tag{5}$$

$$\forall i.\forall \sigma \in [\![\Gamma]\!].\sigma(v_i) \in \sigma([\![t_i]\!]) \qquad \text{(3 and Lemma G.15)} \tag{6}$$

$$\forall \sigma \in [\![\Gamma]\!].\sigma(\phi)[\overline{x_i \mapsto v_i}] \qquad \text{(Lemma G.13, 1, 2, and 3)} \tag{7}$$

According to denotation of Pat and assumption 5, we have

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H\cdot\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle)]\!].\forall \alpha_f \in [\![F]\!].\forall \alpha\ \beta\ \beta'\ e_h\ e_f.$$
$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, \sigma(e)) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \alpha_h +\!\!+ \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \implies$$
$$\alpha \in [\![\sigma(A)]\!]) \qquad \text{(assumption 5)} \tag{8}$$

From now, we consider each $\sigma \in [\![\Gamma]\!]$, and try to prove the subgoal of this case, i.e.,

$$\sigma(\mathbf{gen}\ \mathbf{op}\ \overline{v}\ \mathbf{in}\ e) \in [\![\sigma([H][\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle\cdot A][F])]\!]$$

According to denotation of Pat, we need to prove for all $\forall \alpha_h\ \alpha_f\ \alpha\ \beta\ \beta'\ e_h\ e_f.$ where $\alpha_h \in [\![\sigma(H)]\!]$ and $\alpha_f \in [\![\sigma(F)]\!]$,

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, \mathbf{gen}\ \mathbf{op}\ \overline{\sigma(v_i)}\ \mathbf{in}\ \sigma(e))) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \alpha_h +\!\!+ \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \implies$$
$$\alpha \in [\![\sigma(\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle\cdot A)]\!]$$

Then we have

$$\sigma \in [\![\Gamma]\!] \land \alpha_h \in [\![\sigma(H)]\!] \land \alpha_f \in [\![\sigma(F)]\!] \qquad \text{(assumption)} \tag{9}$$

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \qquad \text{(assumption)} \tag{10}$$

$$\alpha_h \vDash (\beta, \mathbf{gen}\ \mathbf{op}\ \overline{\sigma(v_i)}\ \mathbf{in}\ \sigma(e))) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \qquad \text{(assumption)} \tag{11}$$

$$\alpha_h +\!\!+ \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \qquad \text{(assumption)} \tag{12}$$

$$[\mathbf{op}(\overline{\sigma(v_i)})] \in [\![\mathcal{S}\langle \mathbf{op} \mid \sigma(\phi)[\overline{x_i \mapsto v_i}]\rangle]\!] \qquad \text{(lemma G.4)} \tag{13}$$

$$\alpha_h +\!\!+ [\mathbf{op}(\overline{\sigma(v_i)})] \in [\![\sigma(H\cdot\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle)]\!] \qquad \text{(lemma G.5, 12, and 11)} \tag{14}$$

$$\exists \alpha'.\alpha = \mathbf{op}(\overline{\sigma(v_i)}) :: \alpha' \land \alpha_h \models \mathbf{op}(\overline{\sigma(v_i)}) \Downarrow \beta_{\mathbf{op}} \land$$
$$\alpha_h +\!\!+ [\mathbf{op}(\overline{\sigma(v_i)})] \vDash (\beta \cup \beta_{\mathbf{op}}, \sigma(e)) \xhookrightarrow{\alpha'}{}^{*} (\beta', ()) \qquad \text{(StGen and 12)} \tag{15}$$

Now, we can apply hypothesis 8 with

$$\sigma \mapsto \sigma \quad \alpha_h \mapsto \alpha_h +\!\!+ [\mathbf{op}(\overline{\sigma(v_i)})] \quad \alpha_f \mapsto \alpha_f \quad \alpha \mapsto \alpha' \quad \beta \mapsto \beta \cup \beta_{\mathbf{op}} \quad e_h \mapsto e_h; \mathbf{gen}\ \mathbf{op}\ \overline{\sigma(v_i)}\ \mathbf{in}\ () \quad e_f \mapsto e_f$$

Then we have

$$\alpha' \in [\![\sigma(A)]\!] \qquad \text{(hypothesis 8 with 9, 11, 12, 15, 16)} \tag{16}$$

$$[\mathbf{op}(\overline{\sigma(v_i)})] +\!\!+ \alpha' \in [\![\sigma(\mathcal{S}\langle \mathbf{op} \mid \phi[\overline{x \mapsto v}]\rangle\cdot A)]\!] \qquad \text{(hypothesis 16)} \tag{17}$$

that is sufficient to prove subgoal of this case.

$$\text{Case}\ : \quad \frac{\begin{array}{c}\Delta(\mathbf{op}) = \langle \mathbf{obs}\ \tau, \Theta' \rangle \\ \Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A\cdot F] \\ \Gamma, \overline{x{:}t}; \Delta; \Theta \cup \Theta' \vdash e : [H\cdot\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A][F]\end{array}}{\Gamma; \Delta; \{\mathbf{op}\} \cup \Theta \vdash \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathbf{op} \mid \phi\rangle\cdot A][F]}\ \text{TObs}$$

This rule assume that $e \equiv \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e$, $\tau \equiv [H][\mathcal{S}\langle \mathbf{op} \mid \phi\rangle\cdot A][F]$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(\mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathbf{op}\ \mathbf{in}\ e) \in [\![\sigma([H][\mathcal{S}\langle \mathbf{op} \mid \phi\rangle\cdot A][F])]\!]$$

From the induction hypothesis and the precondition of this rule, we have

$$\Delta(\mathsf{op}) = \langle \textbf{obs}\ \tau, \Theta' \rangle \qquad \text{(assumption)} \tag{1}$$

$$\Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A{\cdot}F] \qquad \text{(assumption)} \tag{2}$$

$$\Gamma, \overline{x{:}t}; \Delta; \Theta \cup \Theta' \vdash e : [H{\cdot}\mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A][F] \qquad \text{(assumption)} \tag{3}$$

$$\forall \sigma \in [\![\Gamma, \overline{x{:}t}]\!].\sigma(e) \in [\![\sigma([H{\cdot}\mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A][F])]\!] \qquad \text{(induction hypothesis)} \tag{4}$$

According to denotation of Pat and assumption 4, we have

$$\forall \sigma \in [\![\Gamma, \overline{x{:}t}]\!].\forall \alpha_h \in [\![\sigma(H{\cdot}\mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle)]\!].\forall \alpha_f \in [\![F]\!].\forall \alpha\ \beta\ \beta'\ e_h\ e_f.$$

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, \sigma(e)) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \alpha_h {+\!\!+} \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \Longrightarrow$$

$$\alpha \in [\![\sigma(A)]\!]) \qquad \text{(assumption 4)} \tag{5}$$

From now, we consider each $\sigma \in [\![\Gamma]\!]$, and try to prove the subgoal of this case, i.e.,

$$\sigma(\textbf{let}\ \overline{x} = \textbf{obs}\ \mathsf{op}\ \textbf{in}\ e) \in [\![\sigma([H][\mathcal{S}\langle \mathsf{op} \mid \phi\rangle{\cdot}A][F])]\!]$$

According to denotation of Pat, we need to prove for all $\forall \alpha_h\ \alpha_f\ \alpha\ \beta\ \beta'\ e_h\ e_f.$ where $\alpha_h \in [\![\sigma(H)]\!]$ and $\alpha_f \in [\![\sigma(F)]\!]$,

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, \textbf{let}\ \overline{x} = \textbf{obs}\ \mathsf{op}\ \textbf{in}\ \sigma(e)) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \alpha_h {+\!\!+} \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \Longrightarrow$$

$$\alpha \in [\![\sigma(\mathcal{S}\langle \mathsf{op} \mid \phi\rangle{\cdot}A)]\!]$$

Then we have

$$\sigma \in [\![\Gamma]\!] \land \alpha_h \in [\![\sigma(H)]\!] \land \alpha_f \in [\![\sigma(F)]\!] \qquad \text{(assumption)} \tag{6}$$

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \qquad \text{(assumption)} \tag{7}$$

$$\alpha_h \vDash (\beta, \textbf{let}\ \overline{x} = \textbf{obs}\ \mathsf{op}\ \textbf{in}\ \sigma(e)) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \qquad \text{(assumption)} \tag{8}$$

$$\alpha_h {+\!\!+} \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \qquad \text{(assumption)} \tag{9}$$

$$\exists \alpha'.\exists \overline{c_i}.\alpha = \mathsf{op}(\overline{c_i}) :: \alpha' \land \alpha_h \models \mathsf{op}(\overline{c_i}) \Downarrow \beta_{\mathsf{op}} \land$$

$$\alpha_h {+\!\!+} [\mathsf{op}(\overline{c_i})] \vDash (\beta \cup \beta_{\mathsf{op}}, \sigma(e[\overline{x_i \mapsto c_i}])) \xhookrightarrow{\alpha'}{}^{*} (\beta', ()) \qquad \text{(StObs and 8)} \tag{10}$$

$$[\mathsf{op}(\overline{c_i})] \in [\![\langle \mathsf{op}\ \overline{y} \mid \sigma(\phi) \land \overline{y = c_i}\rangle]\!] \qquad \text{(lemma G.4, and } \overline{y} \cap \mathtt{DOM}(\Gamma) = \emptyset) \tag{11}$$

$$\sigma([\mathsf{op}(\overline{x})])[\overline{x \mapsto c}] \in \sigma([\![\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle]\!])[\overline{x \mapsto c}] \qquad \text{(lift a new substitution } [\overline{x \mapsto c}] \text{ from 11)} \tag{12}$$

$$\sigma(\alpha_h {+\!\!+} [\mathsf{op}(\overline{x})])[\overline{x \mapsto c}] \in \sigma([\![H{\cdot}\mathcal{S}\langle \mathsf{op} \mid \phi \land \overline{y = x}\rangle)]\!])[\overline{x \mapsto c}] \qquad \text{(lemma G.5, 5, and 12)} \tag{13}$$

Now, we can apply hypothesis 5 with

$$\sigma \mapsto \sigma[\overline{x \mapsto c}] \quad \alpha_h \mapsto \alpha_h {+\!\!+} [\mathsf{op}(\overline{x})] \quad \alpha_f \mapsto \alpha_f \quad \alpha \mapsto \alpha' \quad \beta \mapsto \beta \cup \beta_{\mathsf{op}} \quad e_h \mapsto e_h; \textbf{let}\ \overline{x} = \textbf{obs}\ \mathsf{op}\ \textbf{in}\ \sigma(e) \quad e_f \mapsto e_f$$

Then we have

$$\alpha' \in [\![\sigma(A[\overline{x \mapsto c}])]\!] \qquad \text{(hypothesis 5 with 6, 7, 9, 13)} \tag{14}$$

$$\alpha' \in [\![\sigma(A)]\!] \qquad (A \text{ is well formed under context } \Gamma \text{ and 14}) \tag{15}$$

$$[\mathsf{op}(\overline{c_i})] {+\!\!+} \alpha' \in [\![\sigma(\mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x \mapsto v}]\rangle{\cdot}A)]\!] \qquad \text{(hypothesis 15)} \tag{16}$$

that is sufficient to prove subgoal of this case.

Case : $\dfrac{}{\Gamma; \Delta; \emptyset \vdash () : [H][\square\langle\bot\rangle][F]}$ TRet

This rule assume that $\Theta \equiv \emptyset$, $e \equiv ()$, $\tau \equiv [H][\square\langle\bot\rangle][F]$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \Longrightarrow \sigma(()) \in [\![\sigma([H][\square\langle\bot\rangle][F])]\!]$$

that is, prove the term () is in the denotation of a Pat in from $[H][\square\langle\bot\rangle][F]$. According to the definition of Pat denotation, for all$\alpha_h\ \alpha\ \alpha_f\ \beta\ \beta'\ e_h\ e_f$, where $\alpha_h \in [\![H]\!] \land \alpha_f \in [\![F]\!]$, we need to show

$$[\,] \vDash (\emptyset, e_h) \xhookrightarrow{\alpha_h}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, ()) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \alpha_h {+\!\!+} \alpha \vDash (\beta', e_f) \xhookrightarrow{\alpha_f}{}^{*} (\emptyset, ()) \Longrightarrow \alpha \in [\![\square\langle\bot\rangle]\!]$$

Since there is no small-step reduction rule for the term $()$, thus the relation $\alpha_h \vDash (\beta, ()) \overset{\alpha}{\hookrightarrow}^* (\beta', ())$ is derived from reflexivity case of multi-step reduction. Thus, α is empty trace $[]$, which included by the denotation of $\square\langle\bot\rangle$ (Theorem G.2). Then the proof immediate holds in this case.

Case : $\dfrac{\Gamma;\Delta;\Theta \vdash e_1 : \tau \quad \Gamma;\Delta;\Theta \vdash e_2 : \tau}{\Gamma;\Delta;\Theta \vdash e_1 \oplus e_2 : \tau}$ TChoice

This rule assume that $e \equiv e_1 \oplus e_2$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(e_1 \oplus e_2) \in [\![\sigma(\tau)]\!]$$

From the inductive hypothesis of this case, we know

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(e_1) \in [\![\sigma(\tau)]\!]$$
$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(e_2) \in [\![\sigma(\tau)]\!]$$

Then the Lemma G.6 is sufficient to prove the subgoal of this case.

Case : $\dfrac{\Gamma, z{:}\{\nu{:}\mathtt{unit} \mid \phi\};\Delta;\Theta \vdash e : \tau \quad z \text{ is fresh}}{\Gamma;\Delta;\Theta \vdash \textbf{assume } \phi \textbf{ in } e : \tau}$ TAssume

This rule assume that $e \equiv \textbf{assume } \phi \textbf{ in } e$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(\textbf{assume } \phi \textbf{ in } e) \in [\![\sigma(\tau)]\!]$$

From the inductive hypothesis of this case, we know

$$\forall \sigma, \sigma \in [\![\Gamma, z{:}\{\nu{:}\mathtt{unit} \mid \phi\}]\!] \implies \sigma(e) \in [\![\sigma(\tau)]\!]$$

Since z is a fresh variable, then we have

$$\forall \sigma, \sigma \in [\![\Gamma, z{:}\{\nu{:}\mathtt{unit} \mid \phi\}]\!] \implies \exists \sigma'.\sigma'[\overline{z \mapsto ()}] = \sigma.\sigma'(e) \in [\![\sigma'(\tau)]\!]$$

Moreover, according to the definition of type context denotation,

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \land \sigma(\phi) \iff \sigma[\overline{z \mapsto ()}] \in [\![\Gamma, z{:}\{\nu{:}\mathtt{unit} \mid \phi\}]\!]$$

Then it is safe to apply Lemma G.7 with σ as substitution in $[\![\Gamma]\!]$ and make $\sigma(\phi)$ holds, and $e \mapsto \sigma(\textbf{assume } \phi \textbf{ in } e), e' \mapsto \sigma(e), \tau \mapsto \sigma(\tau)$. Now, we need to show $\textbf{assume } \phi \textbf{ in } e$ can reduced into e without add new effect, which is can be proved by StAssume and $\sigma(\phi)$. Then the proof immediate holds in this case.

Case : $\dfrac{\Gamma;\Delta;\Theta \vdash e : \tau \quad \Gamma \vdash () : \{\nu{:}\mathtt{unit} \mid \phi\}}{\Gamma;\Delta;\Theta \vdash \textbf{assert } \phi \textbf{ in } e : \tau}$ TAssert

This rule assume that $e \equiv \textbf{assert } \phi \textbf{ in } e$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(\textbf{assert } \phi \textbf{ in } e) \in [\![\sigma(\tau)]\!]$$

From the assumption and inductive hypothesis of this case, we know

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(e) \in [\![\sigma(\tau)]\!] \land \sigma(\phi)$$

Then it is safe to apply Lemma G.7 with $e \mapsto \sigma(\textbf{assert } \phi \textbf{ in } e), e' \mapsto \sigma(e), \tau \mapsto \sigma(\tau)$. Now, we need to show $\textbf{assert } \phi \textbf{ in } e$ can reduced into e without add new effect, which is can be proved by StAssert and $\sigma(\phi)$. Then the proof immediate holds in this case.

Case : $\dfrac{\Gamma \vdash op : t \quad \Gamma \vdash t <: \overline{y{:}t} \to t_x \; \forall i.\Gamma \vdash v_i : t_i \quad \Gamma, x{:}t_x[\overline{y \mapsto v}];\Delta;\Theta \vdash e : \tau}{\Gamma;\Delta;\Theta \vdash \textbf{let } x{:}b = op\, \overline{v} \textbf{ in } e : \tau}$ TOpApp

This rule assume that $e \equiv \textbf{let } x{:}b = op\, \overline{v} \textbf{ in } e$, thus we need to prove

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \implies \sigma(\textbf{let } x{:}b = op\, \overline{v} \textbf{ in } e) \in [\![\sigma(\tau)]\!]$$

From the assumption and inductive hypothesis of this case, we know

$$\begin{array}{lll}
\Delta(\textbf{op}) = t & \text{(assumption)} & (1)\\
\Gamma \vdash t <: \overline{y{:}t} \to t_x & \text{(assumption)} & (2)\\
\forall i.\Gamma \vdash v_i : t_i & \text{(assumption)} & (3)\\
\Gamma, x{:}t_x[\overline{y \mapsto v}]; \Delta; \Theta \vdash e : \tau & \text{(assumption)} & (4)\\
\forall v_x.\Gamma \vdash v_x : t_x[\overline{y \mapsto v}] \implies \Gamma; \Delta; \Theta \vdash e[x \mapsto v_x] : \tau[x \mapsto v_x] & \text{(Lemma G.11 and 4)} & (5)\\
\forall \sigma \in [\![\Gamma]\!].\forall v_x \in [\![\sigma(t_x[\overline{y \mapsto v}])]\!].\sigma(e[x \mapsto v_x]) \in [\![\sigma(\tau[x \mapsto v_x])]\!] & \text{(induction hypothesis and 5)} & (6)\\
\forall i.\forall \sigma \in [\![\Gamma]\!].\sigma(v_i) \in [\![\sigma(t_i)]\!] & \text{(Lemma G.15 and 3)} & (7)\\
\forall \sigma \in [\![\Gamma]\!].\forall c_x.op(\overline{\sigma(v)}) \Downarrow c_x \implies c_x \in [\![t_x[\overline{y \mapsto v}]]\!] & \text{(Lemma G.14, 1, 2, and 6)} & (8)\\
\forall \sigma \in [\![\Gamma]\!].\forall c_x.op(\overline{\sigma(v)}) \Downarrow c_x \implies \sigma(e[x \mapsto c_x]) \in [\![\sigma(\tau[x \mapsto c_x])]\!] & \text{(6 and 8)} & (9)\\
\forall \sigma \in [\![\Gamma]\!].\forall c_x.op(\overline{\sigma(v)}) \Downarrow c_x \implies \sigma(e[x \mapsto c_x]) \in [\![\sigma(\tau)]\!] & \text{(9 and } \tau \text{ is well-formed under } \Gamma) & (10)
\end{array}$$

Then it is safe to apply Lemma G.7 with $e \mapsto \sigma(\textbf{let}\ x{:}b = op\,\overline{v}\ \textbf{in}\ e), e' \mapsto \sigma(e[x \mapsto c_x]), \tau \mapsto \sigma(\tau)$. Now, we need to show $\textbf{let}\ x{:}b = op\,\overline{v}\ \textbf{in}\ e$ can reduced into $e[x \mapsto c_x]$ without add new effect, which is can be proved by StOp and the assumption $op(\overline{\sigma(v)}) \Downarrow c_x$. Then the proof immediate holds in this case.

Case : $\dfrac{\Gamma; \Delta; \Theta \vdash e : \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma; \Delta; \Theta \vdash e : \tau'}$ TSub

The case can be directly proved by Lemma G.10.

□

Realizability. The second part of fundamental theorem provide guarantee for realizability, i.e., a trace can be produce by execution of well-typed term. We say that a trace *realizes* a buffer $\{\overline{m_i}\}$ when it contains all messages in this buffer, i.e., $\alpha_1 \# [m_1] \# ... [m_n] \# \alpha_{n+1}$. We also generalize this idea to automata.

Definition G.17 (Trace realize buffer). A trace α realizes buffer $\{\overline{m_i}\}$ when it contains all messages in this buffer, i.e., $\alpha = \alpha_1 \# [m_1] \# ... [m_n] \# \alpha_{n+1}$, denoted as $\beta \lesssim \alpha$.

Definition G.18 (Automata realize buffer). A automata F realizes the buffer β iff $\exists \alpha \in [\![F]\!].\beta \lesssim \alpha$, denoted as $\beta \lesssim F$.

We now prove a stronger theorem than the second part of the fundamental theorem, where we additionally require that the message buffer after the execution of a well-typed term can be realized by the prophecy automata of the Pat:

Theorem G.19 (Realizability). *Given a well-formed handler specification Δ, A well typed program e at least realize one trace:*

$$\Gamma; \Delta\ \Theta \vdash e : [H][A][F] \implies$$
$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$$

Proof. We proceed by induction over our type judgment $\Gamma; \Delta; \Theta \vdash e : \tau$, which consists of the following 8 cases:

Case : $\dfrac{\begin{array}{c}\Delta(\textbf{op}) = \langle \textbf{gen}\ \tau, \Theta' \rangle \quad \Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \textbf{op} \mid \phi \rangle][A \cdot F] \\ \forall i.\Gamma \vdash v_i : t_i \quad \Gamma; \Delta; \Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \textbf{op} \mid \phi[\overline{x_i \mapsto v_i}] \rangle][A][F]\end{array}}{\Gamma; \Delta; \Theta \vdash \textbf{gen}\ \textbf{op}\ \overline{v_i}\ \textbf{in}\ e : [H][\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x_i \mapsto v_i}] \rangle \cdot A][F]}$ TGen

This rule assume that $e \equiv \textbf{gen}\ \textbf{op}\ \overline{v}\ \textbf{in}\ e, \tau \equiv [H][\mathcal{S}\langle \textbf{op} \mid \phi[\overline{x \mapsto v}] \rangle \cdot A][F]$, thus we need to prove

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].$$
$$\exists \alpha \in [\![\sigma(\mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle \cdot A)]\!].\exists \beta'.\alpha_h \vDash (\beta, \mathbf{gen}\ \mathsf{op}\ \overline{v}\ \mathbf{in}\ e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the induction hypothesis and the precondition of this rule, we have

$$\Delta(\mathsf{op}) = \langle \mathbf{gen}\ \tau, \Theta' \rangle \qquad \text{(assumption)} \qquad (1)$$
$$\Gamma \vdash \tau <: \overline{x{:}t} \to [H][\mathcal{S}\langle \mathsf{op} \mid \phi \rangle][A \cdot F] \qquad \text{(assumption)} \qquad (2)$$
$$\forall i.\Gamma \vdash v_i : t_i \qquad \text{(assumption)} \qquad (3)$$
$$\Gamma \mid \Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x \mapsto v}]\rangle][A][F] \qquad \text{(assumption)} \qquad (4)$$
$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H \cdot \mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x \mapsto v}]\rangle)]\!].$$
$$\forall \beta \in [\![\Theta \cup \Theta']\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F) \qquad \text{(induction hypothesis)} \qquad (5)$$
$$\forall i.\forall \sigma \in [\![\Gamma]\!].\sigma(v_i) \in \sigma([\![t_i]\!]) \qquad \text{(3 and Lemma G.15)} \qquad (6)$$
$$\forall \sigma \in [\![\Gamma]\!].\sigma(\phi)[\overline{x_i \mapsto v_i}] \qquad \text{(Lemma G.13, 1, 2, and 3)} \qquad (7)$$

From now, we consider each $\sigma \in [\![\Gamma]\!]$, $\beta \in [\![\Theta]\!]$, and $\alpha_h \in [\![H]\!]$ and try to prove the subgoal of this case:

$$\exists \alpha \in [\![\sigma(\mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle \cdot A)]\!].\exists \beta'.\alpha_h \vDash (\beta, \mathbf{gen}\ \mathsf{op}\ \overline{v}\ \mathbf{in}\ e) \overset{\alpha}{\hookrightarrow}{}^{*} (\emptyset, ()) \land \beta' \lesssim \sigma(F)$$

Then we have

$$\sigma \in [\![\Gamma]\!] \land \beta \in [\![\Theta]\!] \land \alpha_h \in [\![\sigma(H)]\!] \qquad \text{(assumption)} \qquad (8)$$
$$[\mathsf{op}(\overline{\sigma(v_i)})] \in [\![\mathcal{S}\langle \mathsf{op} \mid \sigma(\phi)[\overline{x_i \mapsto v_i}]\rangle]\!] \qquad \text{(lemma G.4 and 7)} \qquad (9)$$
$$\alpha_h +\!\!+ [\mathsf{op}(\overline{\sigma(v_i)})] \in [\![\sigma(H \cdot \mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle)]\!] \qquad \text{(lemma G.5, 8, and 9)} \qquad (10)$$

According to the well-formed type context (Lemma G.13), 1, 2, 8, we have

$$\exists \beta_{\mathsf{op}}.\beta_{\mathsf{op}} \land [\![\Theta']\!] \land \beta_{\mathsf{op}} \lesssim \sigma(A \cdot F) \land \alpha_h \vDash \mathsf{op}(\overline{c}) \Downarrow \beta_{\mathsf{op}} \qquad \text{(Lemma G.13)} \qquad (11)$$
$$\beta \cup \beta_{\mathsf{op}} \in [\![\Theta \cup \Theta']\!] \qquad \text{(Lemma G.8 and 11)} \qquad (12)$$

Now, we can apply hypothesis 5 with

$$\sigma \mapsto \sigma \quad \alpha_h \mapsto \alpha_h +\!\!+ [\mathsf{op}(\overline{\sigma(v_i)})] \quad \beta \mapsto \beta \cup \beta_{\mathsf{op}}$$

Then we have

$$\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta \cup \beta_{\mathsf{op}}, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F) \qquad \text{(hypothesis 5 with 8, 10, and 11)} \qquad (13)$$
$$\alpha +\!\!+ [\mathsf{op}(\overline{\sigma(v_i)})] \in [\![\sigma(\mathcal{S}\langle \mathsf{op} \mid \phi[\overline{x_i \mapsto v_i}]\rangle \cdot A)]\!] \qquad \text{(lemma G.5, 9, and 13)} \qquad (14)$$

With help of hypothesis 13 and 14, we can instantiate the existential quantified variables as $\alpha \mapsto [\mathsf{op}(\overline{\sigma(v_i)})] +\!\!+ \alpha, \beta' \mapsto \beta'$, and we need to prove

$$\alpha_h \vDash (\beta, \mathbf{gen}\ \mathsf{op}\ \overline{v}\ \mathbf{in}\ e) \xhookrightarrow{[\mathsf{op}(\overline{\sigma(v_i)})] +\!\!+ \alpha}{}^{*} (\beta', ())$$

where

$$\alpha_h \vDash \mathsf{op}(\overline{c}) \Downarrow \beta_{\mathsf{op}} \qquad \text{(hypothesis 11)} \qquad (15)$$
$$\alpha_h \vDash (\beta \cup \beta_{\mathsf{op}}, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \qquad \text{(hypothesis 13)} \qquad (16)$$
$$\alpha_h \vDash (\beta, \mathbf{gen}\ \mathsf{op}\ \overline{v}\ \mathbf{in}\ e) \xhookrightarrow{[\mathsf{op}(\overline{\sigma(v_i)})] +\!\!+ \alpha}{}^{*} (\beta', ()) \qquad (\textsc{StGen}, 15, \text{and } 16) \qquad (17)$$

which is sufficient to prove the subgoal in this case.

Case :
$$\frac{\begin{array}{c}\Delta(\mathsf{op}) = \langle \mathbf{obs}\ \tau, \Theta' \rangle \\ \Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A \cdot F] \\ \Gamma, \overline{x{:}t}; \Delta; \Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \mathsf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A][F]\end{array}}{\Gamma; \Delta; \{\mathsf{op}\} \cup \Theta \vdash \mathbf{let}\ \overline{x} = \mathbf{obs}\ \mathsf{op}\ \mathbf{in}\ e : [H][\mathcal{S}\langle \mathsf{op} \mid \phi \rangle \cdot A][F]} \ \textsc{TObs}$$

This rule assume that $e \equiv \texttt{let}\ \overline{x} = \texttt{obs}\ \mathbf{op}\ \texttt{in}\ e, \tau \equiv [H][\mathcal{S}\langle \mathbf{op} \mid \phi\rangle \cdot A][F], \Theta \equiv \{\mathbf{op}\} \cup \Theta$, thus we need to prove

$$\forall \sigma \in \llbracket \Gamma \rrbracket . \forall \alpha_h \in \llbracket \sigma(H) \rrbracket . \forall \beta \in \llbracket \{\mathbf{op}\} \cup \Theta \rrbracket .$$
$$\exists \alpha \in \llbracket \sigma(\mathcal{S}\langle \mathbf{op} \mid \phi\rangle \cdot A) \rrbracket . \exists \beta' . \alpha_h \vDash (\beta, \texttt{let}\ \overline{x} = \texttt{obs}\ \mathbf{op}\ \texttt{in}\ e) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the induction hypothesis and the precondition of this rule, we have

$$\Delta(\mathbf{op}) = \langle \texttt{obs}\ \tau, \Theta' \rangle \quad \text{(assumption)} \quad (1)$$
$$\Gamma \vdash \tau <: \overline{x_i{:}t_i} \to [H][\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A \cdot F] \quad \text{(assumption)} \quad (2)$$
$$\Gamma, \overline{x{:}t}; \Delta; \Theta \cup \Theta' \vdash e : [H \cdot \mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle][A][F] \quad \text{(assumption)} \quad (3)$$
$$\forall \sigma \in \llbracket \Gamma, \overline{x{:}t} \rrbracket . \forall \alpha_h \in \llbracket \sigma(H \cdot \mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle) \rrbracket .$$
$$\forall \beta \in \llbracket \Theta \cup \Theta' \rrbracket . \exists \alpha \in \llbracket \sigma(A) \rrbracket . \exists \beta' . \alpha_h \vDash (\beta, e) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F) \quad \text{(induction hypothesis)} \quad (4)$$

From now, we consider each $\sigma[\overline{x_i \mapsto v_i}] \in \llbracket \Gamma, \overline{x{:}t} \rrbracket$, $\beta \cup \mathbf{op}(\overline{\sigma(v_i)}) \in \llbracket \{\mathbf{op}\} \cup \Theta \rrbracket$, and $\alpha_h \in \llbracket H \rrbracket$ and try to prove the subgoal of this case:

$$\exists \alpha \in \llbracket \sigma(\mathcal{S}\langle \mathbf{op} \mid \phi\rangle \cdot A) \rrbracket . \exists \beta' . \alpha_h \vDash (\beta, \texttt{let}\ \overline{x} = \texttt{obs}\ \mathbf{op}\ \texttt{in}\ e) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$$

Then we have

$$\sigma \in \llbracket \Gamma \rrbracket \land \alpha_h \in \llbracket \sigma(H) \rrbracket \land \beta \in \llbracket \Theta \rrbracket \land \forall i . \sigma(v_i) \in \llbracket \sigma(t_i) \rrbracket \quad \text{(assumption)} \quad (5)$$
$$[\mathbf{op}(\overline{\sigma(x)})] \in \llbracket \sigma(\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle) \rrbracket \quad \text{(lemma G.4)} \quad (6)$$
$$\alpha_h \mathbin{+\!\!\!+} [\mathbf{op}(\overline{\sigma(x)})] \in \llbracket \sigma(H \cdot \mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle) \rrbracket \quad \text{(lemma G.5, 5, and 6)} \quad (7)$$

According to the well-formed type context (Lemma G.13), 1, 2, and 7, we have

$$\exists \beta_{\mathbf{op}} . \beta_{\mathbf{op}} \land \llbracket \Theta' \rrbracket \land \beta_{\mathbf{op}} \lesssim \sigma(A \cdot F) \land \alpha_h \vDash \mathbf{op}(\overline{\sigma(v_i)}) \Downarrow \beta_{\mathbf{op}} \quad \text{(Lemma G.13)} \quad (8)$$
$$\beta \cup \beta_{\mathbf{op}} \in \llbracket \Theta \cup \Theta' \rrbracket \quad \text{(Lemma G.8 and 8)} \quad (9)$$

Now, we can apply hypothesis 4 with

$$\sigma \mapsto \sigma[\overline{x_i \mapsto v_i}] \quad \alpha_h \mapsto \alpha_h \mathbin{+\!\!\!+} [\mathbf{op}(\overline{\sigma(v_i)})] \quad \beta \mapsto \beta \cup \beta_{\mathbf{op}}$$

Then we have

$$\exists \alpha \in \llbracket \sigma(A) \rrbracket . \exists \beta' . \alpha_h \vDash (\beta \cup \beta_{\mathbf{op}}, e) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F) \quad \text{(hypothesis 4 with 5, 7, and 9)} \quad (10)$$
$$\alpha \mathbin{+\!\!\!+} [\mathbf{op}(\overline{\sigma(v_i)})] \in \llbracket \sigma(\mathcal{S}\langle \mathbf{op}\ \overline{y} \mid \phi \land \overline{y = x}\rangle \cdot A) \rrbracket \quad \text{(lemma G.5, 6, and 10)} \quad (11)$$

With help of hypothesis 10 and 11, we can instantiate the existential quantified variables as $\beta \mapsto \{\mathbf{op}(\overline{\sigma(v_i)})\} \cup \beta, \alpha \mapsto [\mathbf{op}(\overline{\sigma(v_i)})] \mathbin{+\!\!\!+} \alpha$, and we need to prove

$$\alpha_h \vDash (\beta, \texttt{gen}\ \mathbf{op}\ \overline{v}\ \texttt{in}\ e) \xhookrightarrow{[\mathbf{op}(\overline{\sigma(v_i)})] \mathbin{+\!\!\!+} \alpha}{}^{*} (\beta', ())$$

where

$$\alpha_h \vDash \mathbf{op}(\overline{c}) \Downarrow \beta_{\mathbf{op}} \quad \text{(hypothesis 8)} \quad (12)$$
$$\alpha_h \vDash (\beta \cup \beta_{\mathbf{op}}, e) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \quad \text{(hypothesis 10)} \quad (13)$$
$$\alpha_h \vDash (\beta, \texttt{gen}\ \mathbf{op}\ \overline{v}\ \texttt{in}\ e) \xhookrightarrow{[\mathbf{op}(\overline{\sigma(v_i)})] \mathbin{+\!\!\!+} \alpha}{}^{*} (\beta', ()) \quad \text{(StGen, 12, and 13)} \quad (14)$$

which is sufficient to prove the subgoal in this case.

Case : $\dfrac{}{\Gamma; \Delta; \emptyset \vdash () : [H][\square\langle \bot \rangle][F]}$ TRet

This rule assume that $\Theta \equiv \emptyset, e \equiv (), \tau \equiv [H][\square\langle \bot \rangle][F]$, thus we need to prove

$$\forall \sigma \in \llbracket \Gamma \rrbracket . \forall \alpha_h \in \llbracket \sigma(H) \rrbracket . \forall \beta \in \llbracket \Theta \rrbracket . \exists \alpha \in \llbracket \sigma(\square\langle \bot \rangle) \rrbracket . \exists \beta' . \alpha_h \vDash (\beta, ()) \xhookrightarrow{\alpha}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$$

Note that the denotation of empty capability only contains an empty buffer, also only empty trace $[]$ is in the denotation of $\sigma(\Box\langle\bot\rangle)$. Thus, we can instantiate β' as $\emptyset$ and prove $\alpha_h \vDash (\emptyset, ()) \overset{[]}{\hookrightarrow}{}^* (\emptyset, ())$, which immediate holds.

$$\text{Case}: \frac{\Gamma;\Delta;\Theta \vdash e_1 : [H][A][F] \quad \Gamma;\Delta;\Theta \vdash e_2 : [H][A][F]}{\Gamma;\Delta;\Theta \vdash e_1 \oplus e_2 : [H][A][F]}\ \textsc{TChoice}$$

This rule assumes that $e \equiv e_1 \oplus e_2$, thus we need to prove

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e_1 \oplus e_2) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the inductive hypothesis of this case, we know

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e_1) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

We also know $\alpha_h \vDash (\beta, e_1 \oplus e_2) \overset{[]}{\hookrightarrow}{}^* (\beta, e_1)$ from StChoice, Then it is sufficient to prove the subgoal of this case.

$$\text{Case}: \frac{\Gamma, z{:}\{v{:}\mathtt{unit} \mid \phi\};\Delta;\Theta \vdash e : [H][A][F] \quad z \text{ is fresh}}{\Gamma;\Delta;\Theta \vdash \mathbf{assume}\ \phi\ \mathbf{in}\ e : [H][A][F]}\ \textsc{TAssume}$$

This rule assume that $e \equiv \mathbf{assume}\ \phi\ \mathbf{in}\ e$, thus we need to prove

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, \mathbf{assume}\ \phi\ \mathbf{in}\ e) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the inductive hypothesis of this case, we know

$$\forall \sigma \in [\![\Gamma, z{:}\{v{:}\mathtt{unit} \mid \phi\}]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

Since z is a fresh variable, then we have

$$\forall \sigma, \sigma \in [\![\Gamma, z{:}\{v{:}\mathtt{unit} \mid \phi\}]\!] \implies \exists \sigma'.\sigma'[\overline{z \mapsto ()}] = \sigma.\sigma'(e) \in [\![\sigma'(\tau)]\!]$$

Moreover, according to the definition of type context denotation,

$$\forall \sigma, \sigma \in [\![\Gamma]\!] \land \sigma(\phi) \iff \sigma[\overline{z \mapsto ()}] \in [\![\Gamma, z{:}\{v{:}\mathtt{unit} \mid \phi\}]\!]$$

Now, we just need to show $\mathbf{assume}\ \phi\ \mathbf{in}\ e$ can reduced into e without add new effect, which is can be proved by StAssume and $\sigma(\phi)$. Then the proof immediate holds in this case.

$$\text{Case}: \frac{\Gamma;\Delta;\Theta \vdash e : [H][A][F] \quad \Gamma \vdash () : \{v{:}\mathtt{unit} \mid \phi\}}{\Gamma;\Delta;\Theta \vdash \mathbf{assert}\ \phi\ \mathbf{in}\ e : [H][A][F]}\ \textsc{TAssert}$$

This rule assume that $e \equiv \mathbf{assert}\ \phi\ \mathbf{in}\ e$, thus we need to prove

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, \mathbf{assert}\ \phi\ \mathbf{in}\ e) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the assumption and inductive hypothesis of this case, we know

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

Since $\Gamma \vdash () : \{v{:}\mathtt{unit} \mid \phi\}$, we know $\sigma(\phi)$ holds. Now, we need to show $\mathbf{assert}\ \phi\ \mathbf{in}\ e$ can reduced into e without add new effect, which is can be proved by StAssert and $\sigma(\phi)$. Then the proof immediate holds in this case.

$$\text{Case}: \frac{\Gamma \vdash op : t \quad \Gamma \vdash t <: \overline{y{:}t} \to t_x \ \forall i.\Gamma \vdash v_i : t_i \quad \Gamma, x{:}t_x[\overline{y \mapsto v}];\Delta;\Theta \vdash e : [H][A][F]}{\Gamma;\Delta;\Theta \vdash \mathbf{let}\ x{:}b = op\ \overline{v}\ \mathbf{in}\ e : [H][A][F]}\ \textsc{TOpApp}$$

This rule assume that $e \equiv \mathbf{let}\ x{:}b = op\ \overline{v}\ \mathbf{in}\ e$, thus we need to prove

$$\forall \sigma \in [\![\Gamma]\!].\forall \alpha_h \in [\![\sigma(H)]\!].\forall \beta \in [\![\Theta]\!].\exists \alpha \in [\![\sigma(A)]\!].\exists \beta'.\alpha_h \vDash (\beta, \mathbf{let}\ x{:}b = op\ \overline{v}\ \mathbf{in}\ e) \overset{\alpha}{\hookrightarrow}{}^* (\beta', ()) \land \beta' \lesssim \sigma(F)$$

From the assumption and inductive hypothesis of this case, we know

$$
\begin{array}{lll}
\Delta(\mathbf{op}) = t & \text{(assumption)} & (1)\\
\Gamma \vdash t <: \overline{y{:}t} \to t_x & \text{(assumption)} & (2)\\
\forall i. \Gamma \vdash v_i : t_i & \text{(assumption)} & (3)\\
\Gamma, x{:}t_x[\overline{y \mapsto v}]; \Delta; \Theta \vdash e : \tau & \text{(assumption)} & (4)\\
\forall v_x. \Gamma \vdash v_x : t_x[\overline{y \mapsto v}] \implies \Gamma; \Delta; \Theta \vdash e[x \mapsto v_x] : \tau[x \mapsto v_x] & \text{(Lemma G.11 and 4)} & (5)\\
\forall \sigma \in [\![\Gamma, x{:}t_x[\overline{y \mapsto v}]]\!]. \forall \alpha_h \in [\![\sigma(H)]\!]. \forall \beta \in [\![\Theta]\!]. & & \\
\quad \exists \alpha \in [\![\sigma(A)]\!]. \exists \beta'. \alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F) & \text{(induction hypothesis and 4)} & (6)
\end{array}
$$

This reduction step is pure, thus we can directly instantiate α in subgoal as α and apply hypothesis 6, then which is sufficient to prove this case.

Case : $\dfrac{\Gamma; \Delta; \Theta \vdash e : \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma; \Delta; \Theta \vdash e : \tau'}$ TSub

The case can be directly proved by Lemma G.10.

□

Fundamental Theorem. Now fundamental theorem can be proved with the help of Theorem G.16 and Theorem G.19.

Theorem G.20 (Fundamental Theorem). *A well-typed term, i.e., $\Gamma; \Delta; \Theta \vdash e : [H][A][F]$, generates traces consistent with the* Pat *and can also terminate with the message buffer providing the capability.*

$$\forall \sigma \in [\![\Gamma]\!]. \sigma(e) \in [\![\sigma([H][A][F])]\!] \land \forall \alpha_h \in [\![\sigma(H)]\!]. \forall \beta \in [\![\Theta]\!]. \exists \alpha. \exists \beta'. \alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ())$$

Proof. For $\sigma \in [\![\Gamma]\!]$, the first conjunct $\sigma(e) \in [\![\sigma([H][A][F])]\!]$ can be provided directly via Theorem G.16. Additionally, for $\alpha_h \in [\![\sigma(H)]\!]$ and $\beta \in [\![\Theta]\!]$, Theorem G.16 shows that $\exists \alpha \in [\![\sigma(A)]\!]. \exists \beta'. \alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(F)$, which is sufficient to proved the second conjunct. □

G.3 Type Soundness

The type soundness can be proved by fundamental theorem and realizability.

Theorem G.21 (Type Soundness). *Given a well-formed handler specification Δ, with ghost variables $\overline{x{:}b}$ and a violation property A, a controller e that satisfies $\overline{x{:}\{v{:}b \mid \top\}}; \Delta; \emptyset \vdash e : [\Box\langle\bot\rangle][A][\Box\langle\bot\rangle]$, then e at least realize one trace consistent with A:*

$$\exists \overline{c{:}b}. \exists \alpha. [\,] \vDash (\emptyset, e[\overline{x \mapsto c}]) \overset{\alpha}{\hookrightarrow}{}^{*} (\emptyset, ()) \land \alpha \in [\![A[\overline{x \mapsto c}]]\!]$$

Proof. According to the fundamental theorem, we have

$$\overline{x{:}\,\{v{:}b \mid \top\}}; \Delta; \emptyset \vdash e : [\square\langle\bot\rangle][A][\square\langle\bot\rangle] \quad \text{(assumption)} \quad (1)$$

$$\forall\sigma, \sigma \in [\![\overline{x{:}\,\{v{:}b \mid \top\}}]\!] \implies \sigma(e) \in [\![\sigma([\square\langle\bot\rangle][A][\square\langle\bot\rangle])]\!] \quad \text{(Theorem G.16 and 1)} \quad (2)$$

$$\forall\sigma, \sigma \in [\![\overline{x{:}\,\{v{:}b \mid \top\}}]\!] \iff \exists\overline{c{:}b}.\sigma = [\overline{x \mapsto c}] \quad \text{(definition of } [\![\Gamma]\!] \text{ and 2)} \quad (3)$$

$$\forall\overline{c{:}b}.e[\overline{x \mapsto c}] \in [\![[\square\langle\bot\rangle][A[\overline{x \mapsto c}]][\square\langle\bot\rangle]]\!] \quad \text{(hypothesis 2 and 3)} \quad (4)$$

$$\forall\sigma \in [\![\Gamma]\!].\forall\alpha_h \in [\![\sigma(\square\langle\bot\rangle)]\!].\forall\beta \in [\![\emptyset]\!].$$
$$\exists\alpha \in [\![\sigma(A)]\!].\exists\beta'.\alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \beta' \lesssim \sigma(\square\langle\bot\rangle) \quad \text{(Theorem G.19 and 1)} \quad (5)$$

$$\forall\alpha.\alpha \in [\![\square\langle\bot\rangle]\!] \iff \alpha = [\,] \quad \text{(Lemma G.2)} \quad (6)$$

$$\forall\beta.\beta \in [\![\emptyset]\!] \iff \beta = \emptyset \quad \text{(Definition of capability denotation)} \quad (7)$$

$$\forall\beta.\beta \lesssim \square\langle\bot\rangle \iff \beta = \emptyset \quad \text{(Lemma ??)} \quad (8)$$

$$\exists\overline{c{:}b}.\exists\alpha.[\,] \vDash (\emptyset, e[\overline{x \mapsto c}]) \overset{\alpha}{\hookrightarrow}{}^{*} (\emptyset, ()) \quad \text{(5 with 3, 6, 7, 8)} \quad (9)$$

Then, the α is the trace realized by the term e. Now we just need to prove $\alpha \in [\![A[\overline{x \mapsto c}]]\!]$. Notice that the denotation of empty capability only contains empty buffer, the definition of Pat denotation as shown in Fig. 15 indicates

$$\forall\alpha_h \in [\![\sigma(\square\langle\bot\rangle)]\!].\forall\alpha_f \in [\![\sigma(\square\langle\bot\rangle)]\!].\forall\alpha\ \beta\ \beta'\ e_h\ e_f.$$
$$[\,] \vDash (\emptyset, e_h) \overset{\alpha_h}{\hookrightarrow}{}^{*} (\beta, ()) \land \alpha_h \vDash (\beta, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\beta', ()) \land \alpha_h +\!\!+ \alpha \vDash (\beta', e_f) \overset{\alpha_f}{\hookrightarrow}{}^{*} (\emptyset, ()) \implies \alpha \in [\![A[\overline{x \mapsto c}]]\!]$$

Again, according to Lemma G.2,

$$[\,] \vDash (\emptyset, ()) \overset{[\,]}{\hookrightarrow}{}^{*} (\emptyset, ()) \quad \text{(definition of } \hookrightarrow^{*}) \quad (10)$$

$$\alpha \vDash (\emptyset, ()) \overset{[\,]}{\hookrightarrow}{}^{*} (\emptyset, ()) \quad \text{(definition of } \hookrightarrow^{*}) \quad (11)$$

$$\exists\overline{c{:}b}.\exists\alpha \in [\![\sigma(A[\overline{x \mapsto c}])]\!].[\,] \vDash (\emptyset, e) \overset{\alpha}{\hookrightarrow}{}^{*} (\emptyset, ()) \quad \text{(Denotation of Pat, 9,10, and 11)} \quad (12)$$

This is sufficient to establish the original theorem we aim to prove. □

G.4 Synthesis is Sound

As discussed in Sec. 4, our synthesis algorithm first refines the input violation property into a set of realizable abstract traces, then uses the **TermDerive** function to translate these traces into a controller program. We first prove the soundness of the forward and backward synthesis steps, as well as the soundness of the top-level synthesis loop. Then, with the support of the lemmas introduced in Appendix F, we establish the overall soundness of the synthesis algorithm.

First, we formally define realizability of abstract traces. Here we use a stronger definition which guarantees that all symbolic events in the abstract trace are already realizable, i.e., can be produced by a well-typed term.

Definition G.22. A symbolic event $\langle \textbf{op} \mid \phi\rangle$ in abstract trace Π (i.e., $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f$) is *consistent* with handler context Δ and type context, denoted as $\Gamma; \Delta \vdash_H \langle \textbf{op} \mid \phi\rangle \in \Pi$, iff ϕ is satisfiable and

$$\forall\tau.\Delta(\textbf{op}) = \langle(\textbf{gen}\ \tau, \Theta)\rangle \land \tau = \overline{y{:}b} \dashrightarrow \overline{x{:}t} \to [H][A][F] \implies$$
$$\Gamma, \overline{y{:}\{v{:}b \mid \top\}}, \overline{x{:}t} \vdash [H][A][F] <: [\Pi_h][\langle \textbf{op} \mid \phi\rangle][\Pi_f]$$

Definition G.23 (Forward Realizability). A symbolic event $\langle \textbf{op} \mid \phi\rangle$ in abstract trace Π (i.e., $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f$) is *forward realizable* when all message sent by it are received in the future. We denote forward realizability as $\Delta \vdash_{\mathsf{fw}} \langle \textbf{op} \mid \phi\rangle \in \Pi$, such that

$$\Delta(\textbf{op}) = \langle(\tau, \Theta)\rangle \implies \exists\overline{\langle \textbf{op}_\mathtt{i} \mid \phi_i\rangle}.\Theta = \{\textbf{op}_\mathtt{i}\} \land \Pi_f = \Pi_1{\cdot}\mathcal{S}\langle \textbf{op}_\mathtt{1} \mid \phi_1\rangle\ldots{\cdot}\langle \textbf{op}_\mathtt{n} \mid \phi_n\rangle{\cdot}\Pi_n$$

Definition G.24 (Backward Realizability). A symbolic event $\langle \mathbf{op} \mid \phi \rangle$ in abstract trace Π (i.e., $\Pi = \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f$) is *backward realizable* when it is an generatable event or there is a previous event who provide capability to received this event. We denote backward realizability as $\Delta \vdash_{\mathsf{bw}} \langle \mathbf{op} \mid \phi \rangle \in \Pi$, such that

$$\begin{aligned}&\Delta(\mathbf{op}) = \langle (\mathbf{gen}\ \tau, \Theta) \rangle \vee \\ &\Delta(\mathbf{op}) = \langle (\mathbf{obs}\ \tau, \Theta) \rangle \implies \exists \langle \mathbf{op'} \mid \phi' \rangle\ \tau\ \Theta'. \Pi_h = \Pi' \cdot \mathcal{S}\langle \mathbf{op'} \mid \phi' \rangle \cdot \Pi'' \wedge \Delta(\mathbf{op'}) = \langle (\tau, \{\mathbf{op}\} \cup \Theta') \rangle\end{aligned}$$

Definition G.25 (Abstract Trace Realizability). An abstract trace Π is *realizable* under given well-formed handler context Δ, denoted as $\Gamma; \Delta \vdash_R \Pi$, iff all symbolic events in Π are consistent with Δ, and are both forward and backward realizable.

Lemma G.26. *For given well-formed handler context Δ, and type context Γ, and abstract trace Π, we have*

$$\Gamma; \Delta \vdash_R \Pi \implies \exists e. \Gamma; \Delta; \emptyset \vdash e : [\Box\langle\bot\rangle][\Pi][\Box\langle\bot\rangle]$$

Before proving the soundness of forward and backward synthesis, we define a relation that helps map effect operators in set (e.g., $\Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}$) back to locations in abstract trace.

Definition G.27 (Realizable set within abstract trace). A set of effect operator Θ in abstract trace Π is realizable under handler context Δ when all all symbolic events with operator in set Θ are consistent with Δ (forward realizable, backward realizable, resp.), denoted as $\Gamma; \Delta \vdash_H \Theta \subseteq \Pi$ ($\vdash_{\mathsf{fw}}$, $\vdash_{\mathsf{bw}}$, resp.).

Now we prove all input and output of both forward and backward synthesis preserve an invariant, such that all symbolic events with operators in fw (bw, resp.) are forward (backward, resp.) realizable. Moreover, all symbolic events whose operators are in the intersection of these two sets are consistent with handler context.

Definition G.28 (Realizability Invariant). Given a handler context Δ, a 6-tuple $(\Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f)$ satisfies the realizability invariant I_R iff

$$\Delta \vdash_{\mathsf{fw}} \Theta_{\mathsf{fw}} \subseteq \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f \wedge \Delta \vdash_{\mathsf{bw}} \Theta_{\mathsf{bw}} \subseteq \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f \wedge \Gamma; \Delta \vdash_H (\Theta_{\mathsf{bw}} \cup \Theta_{\mathsf{fw}}) \subseteq \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f$$

Lemma G.29 (Forward Synthesis is Sound). *For given well-formed handler context Δ, and type context Γ, and abstract trace Π, the forward synthesis preserves the realizability invariant I_R, moreover,*

$$\begin{aligned}&\mathtt{Forward}(\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f) = (\Gamma', \Theta'_{\mathsf{fw}}, \Theta'_{\mathsf{bw}}, \Pi'_h, \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle, \Pi'_f) \implies \\ &\quad (\forall \sigma. \sigma \in [\![\Gamma']\!] \implies \sigma \in [\![\Gamma]\!]) \wedge (\Gamma' \vdash \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f \subseteq \Pi'_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle \cdot \Pi'_f) \wedge \mathbf{op} \in \Theta'_{\mathsf{fw}}\end{aligned}$$

Proof. `Forward` functions just add new variable bindings into the type context on line 3, so $\forall \sigma. \sigma \in [\![\Gamma']\!] \implies \sigma \in [\![\Gamma]\!]$ holds. Since `Forward` functions perform a piecewise automata conjunction on line 4 - 6, thus $\Gamma' \vdash \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f \subseteq \Pi'_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle \cdot \Pi'_f$ also holds. Moreover, $\mathbf{op} \in \Theta'_{\mathsf{fw}}$ directly satisfied on line 8. Finally, According to Definition G.22, Definition G.23, Definition G.24, `Forward` functions merge the type of $\mathbf{op}$ in Δ (line 2), also pass the non-emptiness check (line 7), which preserves realizability invariant. □

Lemma G.30 (Backward Synthesis is Sound). *For given well-formed handler context Δ, and type context Γ, and abstract trace Π, the forward synthesis preserve invariant I_R, moreover,*

$$\begin{aligned}&\mathtt{Backward}(\Delta, \Gamma, \Theta_{\mathsf{fw}}, \Theta_{\mathsf{bw}}, \Pi_h, \mathcal{S}\langle \mathbf{op} \mid \phi \rangle, \Pi_f) = (\Gamma', \Theta'_{\mathsf{fw}}, \Theta'_{\mathsf{bw}}, \Pi'_h, \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle, \Pi'_f) \implies \\ &\quad (\forall \sigma. \sigma \in [\![\Gamma']\!] \implies \sigma \in [\![\Gamma]\!]) \wedge (\Gamma' \vdash \Pi_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi \rangle \cdot \Pi_f \subseteq \Pi'_h \cdot \mathcal{S}\langle \mathbf{op} \mid \phi' \rangle \cdot \Pi'_f) \wedge \mathbf{op} \in \Theta'_{\mathsf{bw}}\end{aligned}$$

Proof. Since `Backward` functions just add new variable bindings into the type context on line 3, $\forall\sigma.\sigma \in [\![\Gamma']\!] \implies \sigma \in [\![\Gamma]\!]$ holds. Since `Backward` functions perform a piecewise automata conjunction on line 4 - 6, thus $\Gamma' \vdash \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f \subseteq \Pi'_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi'\rangle{\cdot}\Pi'_f$ also holds. Moreover, $\textbf{op} \in \Theta'_{\mathsf{bw}}$ directly satisfied on line 8. Finally, According to Definition G.22, Definition G.23, Definition G.24, `Forward` functions merge the type of $\textbf{op}_{\textbf{parent}}$ in Δ (line 2) which provide capability includes **op**, also pass the non-emptiness check (line 7), which preserves realizability invariant. □

Theorem G.31 (Synthesis is Sound). *The controller synthesized by the algorithm is type-safe with respect to our declarative typing rules.*

Proof. We first show the top-level refinement loop always terminates with a realizable abstract trace. We prove this by contradiction: if this abstract trace Π is not realizable, then according to Definition G.25, there must exist a symbolic event within this abstract trace that is realizable. It can be either not forward realizable, or backward realizable, or consistent with the handler context.

(1) If $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f$ where $\Gamma; \Delta \nvdash_{\mathsf{fw}} \langle \textbf{op} \mid \phi\rangle \in \Pi$, then according to Lemma G.29 and realizable invariant, it cannot be included in Θ_{fw}. Then, the refinement loop will not stop since the condition on line 3 still hold. Moreover, since $\textbf{op} \notin \Theta_{\mathsf{fw}}$ (line 4), the refinement loop will perform forward synthesis, which add **op** into Θ_{fw} (Lemma G.29). This makes $\textbf{op} \in \Theta_{\mathsf{fw}}$, which leads a contradiction.

(2) If $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f$ where $\Gamma; \Delta \nvdash_{\mathsf{bw}} \langle \textbf{op} \mid \phi\rangle \in \Pi$ and **op** is an observable operator, then according to Lemma G.29 and realizable invariant, it cannot be included in Θ_{bw}. Then, the refinement loop will not stop since the condition on line 3 still hold. Moreover, since $\textbf{op} \notin \Theta_{\mathsf{bw}}$ and not generatable (line 6 - 7), the refinement loop will perform backward synthesis, which add **op** into Θ_{bw} (Lemma G.30). This makes $\textbf{op} \in \Theta_{\mathsf{bw}}$, which leads a contradiction.

(3) If $\Pi = \Pi_h{\cdot}\mathcal{S}\langle \textbf{op} \mid \phi\rangle{\cdot}\Pi_f$ where $\Gamma; \Delta \nvdash_{H} \langle \textbf{op} \mid \phi\rangle \in \Pi$. According to the realizability invariant, it cannot be included in Θ_{bw} and it cannot be included in $\Theta_{\mathsf{fw}} \cap \Theta_{\mathsf{bw}}$. Thus, the refinement loop will not stop since the condition on line 3 still hold, which leads a contradiction.

From this argument, it follows that the refined abstract trace Π is realizable under refine type context Γ on line 10. Then Lemma G.26 shows that there exists a term e, such that $\Gamma; \Delta; \emptyset \vdash e : [\Box\langle\bot\rangle][\Pi][\Box\langle\bot\rangle]$. According to the soundness of term derivation (Lemma F.3), we have shown that the synthesized controller e is well-typed. □

H Evaluation Details

Table 2 lists the details for the benchmarks used in our evaluation section. The complete benchmark suite and source code of Clouseau are available at the following anonymous link: https://anonymous.4open.science/r/PLDI25-submission-sp-7D3E

A Docker image is also provided on Zenodo: https://zenodo.org/records/14166141

Table 2. Detail explanation of benchmarks.

Benchmark	Model description	Property to be violated	synthetic fault injected into models
Database	The simplified database used as the running example in Sec. 2	**RYW**: Read-Your-Writes policy described in Sec. 2	Remove atomicity check
EspressoMachine	The user interacts with a coffee machine through its control panel, where the panel must correctly interpret user inputs and handler errors from coffee machine.	**Strong Consistency**: The user, panel, and underline coffee machine should have a consistent view of the state of the machine. Precisely, if the coffee machine is in a state "run out of water", the user should get notification.	Remove error forwarding in panel machine
Simplified2PC	A simplified version of a two-phase commit protocol (2PC), where we assume transactions have a single update operation.	**RYW**: Read-Your-Writes policy	Original implementation doesn't guarantee RYW
HeartBeat	A failure detector that sends heartbeat messages to a node to make sure it is alive; it reports an error only when the node doesn't reply for multiple rounds, taking into account network packet lost.	**Eventual Consistency**: The node and detector should have the same view of state of node (alive or crashed) eventually. Precisely, if the node is alive, the detector will not report a false positive error.	Specify a timer-based protocol that can cause a false positive error.
BankServer	The user interacts with a bank to withdraw money from their accounts, where the balance is stored in another database component.	**Strong Consistency**: The bank and underline store should have consistent view of balance of accounts. Precisely, bank should disallows users from withdraw an amount greater than their current balance.	remove negative balance check in bank machine
RingLeaderElection	Ring election algorithm where a group nodes are interconnected in a ring-like structure.	**Unique Leader Policy:** there can only be a unique node that announces itself as leader.	Omit a node comparison equality check
Firewall	A set of internal and external nodes communicating through a firewall. Firewall should block message from an external node, unless this node has received message from internal nodes previously. The firewall actually keep a whitelist of external nodes that can communicate with internal nodes.	**Liveness:** if an internal node sends a message to an external node, it will eventually be able communicate with an external node.	modify the whitelist updating logic.
ChainReplication	Chain replication protocol[38].	**RYW:** Read-Your-Writes policy.	Remove log recovery logic after node crash
Paxos	Paxos protocol[23].	**Unique Leader Policy:** there are multiple proposers accepted as leaders. This will additionally violates the Paxos agreement policy, i.e., two distinct learners cannot learn different values.	A wrong node comparison in leader election logic
Raft	Raft algorithm[31].	**Strong Consistency:** The leader's view should align with committed data, i.e., if a log entry is committed, then it should also be present in the leader's log.	Incorrect log recovery logic after node crash
Anno2PCModel	Case study in Sec. 5	**Strong Consistency:** the user and the database should have the same will view of stored data, as explained in the case study in Sec. 5	Omit buffered transaction update logic