University of Glasgow | School of Computing Science

Time For Typestates

Luke Gall (2298070)

April 7, 2022

ABSTRACT

Typestates *typecheck the set of operations an object can perform depending on its state. By using typestates, programmers can define and verify the correct ordering of method calls to capture protocols for individual classes.* ***Timed Automata*** *can be used to model real time systems or protocols that are reliant on the correct timing of actions.*

In this work, we present an extension to a typestate typing system which can capture and verify timing constraints on method calls. Our typing system allows typestate method definitions to be annotated with time constraints. The typing system can then capture the time various method calls occur, ensuring they are within their declared timing constraints. We also provide two extensions to existing typestate tools: Timed Mungo and Timed StMungo. Timed Mungo allows users to attach time constrained typestate definitions to a Java class, which can verify classes conform to their time constrained protocols. Timed StMungo can convert timed local session types and convert to a timed typestate definition. Progress and safety is proved for our timed typestate theory and evaluation is performed on a timed SMTP (Simple Mail Transfer Protocol) client.

1. INTRODUCTION

Multiparty Session Types (MPST) [21] codify communication protocols for two or more parties. These session types define the allowed ordering and type of data that can be sent between parties. This theory provides a basis for *static type checking* of communication protocols in a variety of settings. These checks can guarantee mathematical safety for various properties and reduce the likelihood of runtime errors, which can be costly and time-consuming to fix. Mungo [24], a *Typestate* toolchain based on MPST, allows static type checking of communication protocols in Java. This tool allows developers to attach typestate definitions to a class which act as a state machine which can capture the allowed method calls in various states. This allows communication protocols to be captured and statically typed checked.

On a parallel line of research, Timed Automata [5] presented an automaton that captures time constraints for actions that can be used to model a variety of real time systems or protocols. Such protocols include time constraints in the Simple Mail Transfer Protocol (SMTP) such as "An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender." which would require time constraints on the server and client side. Time constraints have been added to a variety of settings [25, 31, 26, 3, 29] including Multiparty Session Types [9].

We present an extension to Mungo and StMungo which draws upon theory from Timed Multiparty Session Types [9] to provide a time typestate theory. *Timed Mungo* allows users to attach timed typestate protocols to a Java class, in which method calls are augmented with time constraints and clocks. A developer can then invoke the *tsDelay* expression to model the passage of time in any setting, including external method calls, or within separate switch branches. Our Timed Mungo system will then ensure that all methods conform to their prescribed time constraints as declared in their typestate definition. Our second extension, Timed StMungo, updates the StMungo tool to translate timed Scribble local types [36] into timed typestates protocols for each local endpoint. Each endpoint protocol can then be attached to Java classes, which can be verified using Timed Mungo.

Our work advances existing work surrounding Mungo [10, 28, 22] by drawing on the parallel line of work involving Timed Multiparty Session Types [9, 29]. As far as we are aware, this tool and its relevant theory is the first example of time constraints for typestates that can be used to verify correct timing of Java programs.

2. BACKGROUND AND RELATED WORK

We provide the background and related work in three distinct areas which act as the base of our research: **Session Types**, **Timed Automata**, and **Typestates**. Our work draws upon these three fields to produce new timed typestate theory.

2.1 Session Types

Background Session types [19, 20] describe a sequence of communications between two or more parties [21]. They can be used to codify communication protocols such that senders and receivers agree on the type of data being sent and the ordering at which it occurs. They can be used to ensure type safety and deadlock freedom in the concurrent setting.

Programming Languages Session types have been added to a variety of calculus [13] and programming languages. One of the first examples of a programming language augmented with session types was *Ldoos* [16] which is an objected-oriented language where channel usages are attached to fields within class definitions following session type syntax. Their work was later updated in a new language, MOOSE [15], which allowed multithreaded (binary) processes to communicate with each other following their defined protocols. Further updates included bounded polymorphism [14] and the amalgamation of session types with method definitions [11]. In this work, classes contain a set of sessions/methods that send asynchronous messages that contain objects on chan-

nels. An overview of session types converted to three programming paradigms was presented by Vasconcelos [34] to provide a clear example of translating session types into three distinct areas. Session types have been provided for the TypeScript programming language [27] which provides communication safety for web development over *WebSockets* using RouST, a new extension of session types that supports communications through routed parties such as servers. *Bica* was presented by Gay et al. [17] which extended Java with session types that could be consumable by multiple methods instead of a single method, seen previously. They extended the existing session type theory for object-oriented languages by allowing session types to be attached to classes instead of individual fields. This session type can specify the sequence of method calls available to different states of the class.

2.2 Time

Background Constraints on timing information were first presented by Timed Automata [5] to model constraints required for real-time systems such as aeroplanes. These automata were created to model a sequence of events that contained constraints based on the time these events occur. These constraints are modelled using *clocks*, which track the ever-increasing global passage of time. These timing constraints were later added to communicating automata to produce *communicating timed automata* (CTA) [25] which can model systems that send and receive messages over channels while following timing constraints. This system builds upon Timed Automata by giving a set of unbound communication channels to a system of timed automata. CTA have been shown to have a sound correspondence to multiparty session types in the untimed setting [7]. Various properties of CTAs were also explored including safety, and progress which required all time constraints to be valid for the system.

Calculus Time constraints have previously been added to π-calculus [31] where individual processes have a unique set of local clocks and actions are augmented with constraints and time-stamps. *Conversation Calculus* [35], which is an extension of π-calculus modelling distributed multiparty mediums, has been adapted to the timed setting [26]. This extension includes time and exception handling, where the user would model default and compensating behaviour. This mode allows an abort mechanism to be created when time progresses past a certain value. Other time related extensions include timed message sequence charts [3] and work to determine if a binary session type admits a *compliant* [6], which is another session type which can be composed to create a sound protocol.

Timed Multiparty Session Types An extension to Multiparty Session Types was created to model time based communication protocols [9]. These Timed Multiparty Session Types allow time constraints to be present on *global* and *local* session types. Each action can have time constraints on the Send and Receive sides. Time error freedom property is proved for this system when a *global type* is *wait-free* and *feasible*. Wait-freedom ensures receive actions that fire never need to wait for the message to be sent by the corresponding sender. The authors used this work to present an update to the Scribble protocol language [36] with time constraints [29]. They then provided an implementation of their work using a timed Python API that demonstrated the practicability of these time constraints using a dynamic runtime monitor.

Asynchronous Timed Session Types Recent work [8] aimed to address the *wait-free* property of timed multiparty session types. This property reduces the expressivity of the system and can fail to capture real-world communication protocols such as SMTP [23]. Their binary asynchronous timed sessions types are updated with *timeout* and *delay* primitives to model time constrained methods, and time passing respectively. They provide subtyping relations on binary timed session types and provide a simulation relation between type configurations. *Time Safety* and *Subject Reduction* are shown for this system.

2.3 Typestates

Background Typestates [32] type the set of operations an object can perform depending on its state. Most objects have implicit state, such as file objects being open or closed. Typestates codify the set of operations that are available in each state and be used to remove method ordering errors. After performing an action, the typestate will progress to a new state.

Programming Languages *Plaid* [4, 33] introduced a new typestate programming paradigm. In this paradigm, objects are intertwined with states and the transitions between them. Plaid aims to introduce typestates within the language syntax rather than a separate checker. Programmers would write the states that a class contains, each with individual methods and fields. This language targets the JVM and supports substates, a subclass like formulation, and supports dynamic type checking. Typestates have also been adapted to a Chemistry metaphor [12] in which states act as molecules of messages. This concurrent system allows processes to change and access different objects in the "chemical soup". Each object can present a different set of methods at different points in time, capturing the features of typestates. Their system was later updated [30] to provide deadlock freedom for typestate oriented programming.

Mungo A typestate toolchain based on multiparty session types was presented by Kouzapas et al. with Mungo and StMungo [24]. Intuitively, session types capture the permitted sequence of communication calls between processes, this can be translated to capture the permitted sequence of method calls an object can perform. Scribble [36] protocols can be translated into typestate definitions which can be attached to Java classes. The typestate definition describes a state machine of valid method calls which can be type checked using their external static type checker. Progress and Type Preservation is provided for their theory and provides a bridge between session types and typestates. Extensions to Mungo include null safety [10], field annotations and droppable states [28], as well as unrestricted aliasing in Scala [22].

3. FORMALISATION

In the following section, we present the calculus for our updated timed typestate theory, which acts as an extension

$$
\begin{array}{lll}
\kappa & ::= & [\delta, P] & \textbf{Time Constraint} \\
\delta & ::= & \mathit{true} \mid \pi > c \mid \pi = c \mid \neg\delta \mid \delta_1 \wedge \delta_2 & \textbf{Clock Constraint} \\
P & ::= & \widetilde{\pi} & \textbf{Reset Predicate} \\
 & & & \\
D & ::= & \mathsf{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \mid \mathsf{enum}\ E\ \{\widetilde{l}\} & \textbf{Type Decl} \\
F & ::= & T\ f & \textbf{Field} \\
M & ::= & T\ m(T\ x)\ \{e\} & \textbf{Method} \\
T & ::= & C \mid E \mid \mathsf{bool} \mid \mathsf{void} & \textbf{Type} \\
c & ::= & l \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{null} \mid * & \textbf{Constant} \\
o & ::= & \mathsf{this} \mid o.f \mid x & \textbf{Path} \\
e & ::= & c \mid o \mid o.m(e) \mid o.f = e \mid e; e & \\
 & & \mid o.f = \mathsf{new}\ C \mid \lambda : e & \\
 & & \mid \mathsf{continue}\ \lambda \mid \mathsf{switch}\ (e)\ \{e_l\}_{l \in E} & \\
 & & \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \mid \mathsf{delay}(t) & \textbf{Expr} \\
 & & & \\
S & ::= & \widetilde{G} \mid \mu X.S \mid X & \textbf{Declared Typestate} \\
G & ::= & \ulcorner\kappa\urcorner\ T\ m(T) : S \mid E\ m(T) : \langle \ulcorner\kappa_l\urcorner\ S_l \rangle_{l \in E} & \textbf{Signature} \\
 & & & \\
I & ::= & \widetilde{B} \mid \mu X.I \mid X & \textbf{Inferred-Typestate} \\
N & ::= & T\ m(T) : I \mid E\ m(T) : \langle I_l \rangle_{l \in E} & \textbf{Inferred-Signature} \\
B & ::= & N \mid (t, I) & \textbf{Inferred-Branch} \\
 & & & \\
\Omega & ::= & C[I] \mid E \mid \mathsf{bool} \mid \mathsf{void} \mid \mathsf{bot} & \textbf{Inferred-Types} \\
\Lambda & ::= & \emptyset \mid \Lambda, o : \Omega \mid \Lambda, \lambda : X & \textbf{Typing Context}
\end{array}
$$

Figure 1: Syntax

to the typestate theory presented in the original Mungo paper [24]. We present our timed typestate calculus, including reduction relations, to define a timed typestate inference system. After describing our inference system, we give some "handwritten" derivations to aid understanding before discussing how inferred typestates are typechecked using our subtyping relation.

3.1 Calculus

We present the syntax of Timed Mungo, a core object-oriented calculus. This calculus is an extension of Mungo presented by Kouzapas et al. [24]. This core calculus is used to formalise the typestate inference system discussed in Section 3.3.

Time We model time using the theory presented in Timed Automata [5]. Let Π be a set of clocks ranging over $\pi_1, \pi_2, ..., \pi_n$, taking values in $\mathbb{R}_{\geq 0}$. $v : \Pi \to \mathbb{R}_{\geq 0}$, is a *clock assignment* which returns the times of clocks in Π. We write $v+t$ assignment mapping all $\pi \in \Pi$ to $v(\pi) + t$. The initial assignment mapping all clocks to 0 is written as v_0. t is a time constant in $\mathbb{Q}_{\geq 0}$. We write $v \models \delta$ when δ is satisfied by v. A reset predicate P contains a subset of Π. When P is $\emptyset$ no resets occur and clocks are untouched, else for each $\pi \in \Pi$, π is set to 0. We write $v[P \to 0]$ for the reset of all clocks in P to 0.

Core Syntax Our syntax presented in Fig. 1 extends that presented in Mungo with time constraints. Items represented using $\widetilde{\cdot}$ represent a possibly empty set. *Type declarations* $\{D\}$ represent a program, which can be a set of either an enum or a class. Class definitions declare a *typestate* specification, and then a possibly empty set of fields, F, and methods, M. An enum type is defined with a non-empty set of label enum values $\{l\}$. Class, enumeration, and method names are assumed to be unique. For any $\mathsf{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \{D\}$, we have fields(C) = $\{F\}$, methods(C) = $\{M\}$, and typestate(C) = S. We have enums(E) = l for any $\mathsf{enum}\ E\ \{\widetilde{l}\}$.

$T\ m(T\ x)\ \{e\}$ is a method declaration which specifies T the return type, m the method name, the parameter x with a type of T', and the method body which is represented by an expression e. Paths denote the current object / receiver by this or with the path $o.f$ which is the field of an object. For *constants*, we have either null, bool literal values tt or ff, * which is void type, and l an enum value.

For *expression*, we have method calls, field assignment, and object creation where the target object is referenced by a path o, this is required for the target's typestate to be tracked. We also have sequences, switch expressions, if else, labelled expressions, continues, and the delay(t) expression. This delay expression acts as $v+t$ for all clocks globally. This ensures time is global, and any method that progresses time also progresses it for all other typestates in the typing environment.

Declared Typestates A *declared typestate* S defines a state machine which contains the methods of a class as its actions. A typestate is either a set of method signatures $\{G\}$ which corresponds to an *internal choice*, a recursive typestate $\mu X.S$, or the recursive variable X. A typestate method signature can either transition to a new typestate S or represents an *external choice* $E\ m(T) : \langle \ulcorner\kappa_l\urcorner\ S_l \rangle_{l \in E}$ where the method call returns an *enum* value which transitions to a unique typestate based on the label, l, returned.

For both types of method signature, we have an optional *time constraint* $\ulcorner\kappa\urcorner$. A Time Constraint contains a clock constraint δ and a reset predicate P. This adds time constraints to *internal* and *external choices*. The individual labelled branches of an external choice are each prefixed with a unique time constraint, in order for a process to be valid against each of these time constraints, they must overlap with their time constraints. Each branch may contain a unique *reset predicate*, required to represent different paths of time progression. Clock constraints, δ, follow the definitions found in Timed Automata, and relate to constraints on the timings of unique clocks held within the typestate. Clocks held by individual typestates are unique and not shared by other objects. For the clock constraints $\mathit{false}, <, \leq, \geq$ and $\vee$ we derive them in the standard way.

Inferred Typestate. The syntax of an inferred typestate if given in Fig. 1. The syntax follows closely to the existing syntax for declared typestate without clock constraints and the addition of inferred branches. An inferred branch can either be a method call, similar to the declared typestate, or of the form (t, I) which is used to capture a delay expression where t represents the time value when I occurs. For example, $(5, T\,m(T) : I)$ infers that the method was called after a *delay*(5) expression. Individual delay expressions each add a new t to the inferred typestate, such as $(4, (3, (2, I)))$. This simplifies later reduction and subtyping rules.

Typing Syntax At the bottom of Fig. 1 we present the syntax for our typing system. *Inferred Types*, Ω, involve a class associated with an *inferred* typestate, I, an enum E, boolean, void, and bottom value used for recursion. Note that classes are only ever inferred with an inferred typestate, instead of the declared subtype S. A clear distinction between the two separate typestates is required to separate typestates that have time constraints and typestates have the notion of time progression.

The transition relation $I \xrightarrow{s} I'$, is defined by the following rules:

$$T\ m(T) : I \xrightarrow{T\ m(T)} I \qquad \frac{l' \in \mathsf{enums}(E)}{E\ m(T) : \langle I_l \rangle_{l \in E} \xrightarrow{E\ m(T);l'} I_{l'}}$$

$$\frac{B \in \widetilde{B} \qquad B \xrightarrow{s} I}{\widetilde{B} \xrightarrow{s} I} \qquad \frac{I\{\mu X.I/X\}) \xrightarrow{s} I'}{\mu X.I \xrightarrow{s} I'} \qquad \frac{l' \in \mathsf{enums}(E)}{\langle I_l \rangle_{l \in E} \xrightarrow{l'} I_{l'}} \qquad (t, I) \xrightarrow{t} I$$

Figure 2: Reduction Rules for inferred Typestates

Our typing context, Λ, contains information about inferred types for paths, o, as well as mapping labels, λ, to their recursion variables X.

3.2 Typestate Reductions

This subsection provides details of reduction relations for declared and inferred typestates which take inspiration from the reduction relations presented in the original Mungo [24]. We define the reduction relations for inferred typestates in Fig. 2. Internal method calls have no premise condition, while external choices can reduce by any valid label. The branches rule allows a set of branches to reduce to the continuation of one of the elements of the set. The recursive rule unfolds the typestate definition. The time rule for the inferred type removes the time information from a typestate. This follows the idea that when time passes we no longer need extra information to convey the time when I occurred as time has "caught up" to this point.

We define a transition relation on declared typestates as shown in Fig. 3. The reduction relation for declared typestates is updated to be associated with a clock assignment function v. This function will store the clocks values required for the time constraints of S. Reductions on internal choice and external choices methods both ensure that a method can only reduce if its clock constraints are valid. After reducing, the new clock assignment is updated via the reset predicate of the function call. The time reduction uses a helper function *rdy* defined in the Appendix as Fig. 14. This function gathers the clock constraints of the available actions of S. It is based on the *rdy* function found within Multiparty Timed Session Types [21]. The premise on the time reduction ensures that time can pass as long as one ready action (method) is valid regarding its clock constraints, when this action is an external choice all its time constraints must be valid. Formally, we have $v \models^* rdy(S)$ iff $\exists \{\delta\} \in rdy(S), \exists t \geq 0. \forall \delta \in \{\delta\}. v + t \models \delta$. For both declared reduction relations above, $\overline{s}$ can be used to represent a sequence of reductions.

3.3 Inferring Typestates

This section will describe the typestate inference system of Mungo [24] and our extensions present in Timed Mungo. The following formulation assumes a method is analysed each time it is called, this is not present in the implementation in order to allow recursive methods. The typestate inference system works by inferring an *inferred typestate* for a class following the static usages of that class. This inferred typestate, I, is then checked against its declared counterpart, S paired with its clock values, v. The inference system makes use of the typing syntax discussed in Section 3.1.

The transition relation $(v, S) \xrightarrow{s} (v', S')$, where v can be assumed to contain the clocks of S, is defined by the following rules:

$$\frac{v \models \delta \qquad v' = v[P \to 0]}{(v, [\delta, P]\ T\ m(T) : S) \xrightarrow{T\ m(T)} (v', S)} \qquad (v, T\ m(T) : S) \xrightarrow{T\ m(T)} (v, S)$$

$$\frac{l' \in \mathsf{enums}(E) \qquad v \models \delta_l \qquad v' = v[P_l \to 0]}{(v,\ E\ m(T) : \langle \ulcorner\ [\delta_l, P_l]\ \urcorner\ S_l \rangle_{l \in E}) \xrightarrow{E\ m(T);l'} (v', S_{l'})}$$

$$\frac{G \in \widetilde{G} \qquad (v, G) \xrightarrow{s} (v', S)}{(v, \widetilde{G}) \xrightarrow{s} (v', S)} \qquad \frac{(v, S\{\mu X.S/X\}) \xrightarrow{s} (v', S')}{(v, \mu X.S) \xrightarrow{s} (v', S')}$$

$$\frac{l' \in \mathsf{enums}(E) \qquad v \models \delta_l \qquad v' = v[P_l \to 0]}{(v, \langle \ulcorner\ [\delta_l, P_l]\ \urcorner\ S_l \rangle_{l \in E}) \xrightarrow{l'} (v', S_{l'})}$$

$$\frac{v' = v + t \qquad v' \models^* rdy(S)}{(v, S) \xrightarrow{t} (v', S)}$$

Figure 3: Reduction Rules for declared Typestates

$$\begin{aligned} \mathsf{init}(C) &= \mathsf{null} \\ \mathsf{init}(E) &= E_{\mathsf{init}} \\ \mathsf{init}(\mathsf{bool}) &= \mathsf{ff} \\ \mathsf{init}(\mathsf{void}) &= * \\ \mathsf{init}(\widetilde{\pi}) &= v \qquad \textit{where } v(\pi) = \begin{cases} 0 & \text{if } \pi \in \{\pi\} \\ \textit{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Figure 4: Init function

The inference system uses a subtyping relation $\leqslant$: discussed in Section 3.5 and a binary operator $\mathsf{join}(\cdot,\cdot)$. For inferred types and typing contexts, we have the below relations.

- The *subtyping* relation ($\leqslant$:) for inferred typestates $\leqslant$: is defined by the rules in Fig. 7.
- The *equivalence* relation ($\eqslantless$) for inferred typestates is shown in the appendix, in Fig. 17.
- The *join* operator $\mathsf{join}(\cdot,\cdot)$ is defined in Fig. 5.

The **init** helper function is defined in 4. This function provides the initial values for different types. $init(\widetilde{\pi})$ is used to create a new clock assignment mapping. It takes a set of clock names, and it produces a new clock function v where every clock name is set to an initial value of 0.

The **Join** helper function can be found in Fig. 5, and is an extension of the same function found within the original Mungo paper [24]. This function is used to computer a common inferred typestate from multiple execution paths. Inferred typestates that share a common method prefix are joined, while other methods are joined with a set union. Typestates that share a common time constant t are joined. This allows branches that share a common time setting to merge.

The inference system and rules are extensions from the original Mungo [24] theory. The original theory will be described below, as well as discussion of our extensions. The form of the judgements for the inference rules are as follows:

$$\mathsf{join}(T\ m(T') : I, \widetilde{B}) = \begin{cases} \{T\ m(T') : I\} \cup \widetilde{B}' & \text{if } \widetilde{B} = \{T\ m(T') : I'\} \\ & \cup \widetilde{B}' \wedge I \coloneqq I' \\ \{T\ m(T') : I\} \cup \widetilde{B} & \text{otherwise} \end{cases}$$

$$\mathsf{join}(E\ m(T) : \langle I_l \rangle_{l \in E}, \widetilde{B}) = \begin{cases} \{E\ m(T) : \langle l : I_l \rangle_{l \in E}\} \cup \widetilde{B}' & \text{if } \widetilde{B} = \{E\ m(T) : \langle I'_l \rangle_{l \in E}\} \\ & \cup \widetilde{B}' \wedge I_l \coloneqq I'_{l_{l \in E}} \\ \{E\ m(T) : \langle I_l \rangle_{l \in E}\} \cup \widetilde{B} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{join}(\{B\} \cup \widetilde{B}, \widetilde{B}') &= \mathsf{join}(\widetilde{B}, \mathsf{join}(B, \widetilde{B}')) \\
\mathsf{join}(\mathsf{end}, \mathsf{end}) &= \mathsf{end} \\
\mathsf{join}(\mu X.I_1, I_2) &= \mathsf{join}(I_1\{\mu X.I_1/X\}, I_2) \\
\mathsf{join}(C[I], C[I']) &= C[\mathsf{join}(I, I')] \\
\mathsf{join}((t, I), (t', I')) &= \begin{cases} (t, \mathsf{join}(I, I')) & \text{if } t = t' \wedge I \coloneqq I' \\ \{(t, I)\} \cup \{(t', I')\} & \text{otherwise} \end{cases} \\
\mathsf{join}(N, (t, I)) &= \{N\} \cup \{(t, I)\} \\
\mathsf{join}(\Omega, \mathsf{bot}) &= \Omega \\
\mathsf{join}(\mathsf{bot}, \Omega) &= \Omega \\
\mathsf{join}(\Omega, \Omega) &= \Omega \qquad (\Omega \in \{E, \mathsf{bool}, \mathsf{void}\}) \\
\mathsf{join}(\Lambda, \Lambda') &= \{r : C[\mathsf{join}(I, I')] \mid r : C[I] \in \Lambda, \\
&\qquad r : C[I'] \in \Lambda'\} \ \cup\ (\Lambda \setminus \Lambda') \cup (\Lambda' \setminus \Lambda)
\end{aligned}$$

Figure 5: Join operator (symmetric recursion rule omitted)

$\Lambda \vdash e : \Omega \dashv \Lambda' \qquad \Lambda \vdash C[S] \qquad \vdash \mathsf{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \qquad \vdash \widetilde{D}$

The first judgement is the typing for expressions. The judgement should be read from right to left, however it also holds for left to right. It can be read as taking an input context Λ' and expression e, which then computes the type Ω. The conclusion of Λ' on e is then captured as "output" in Λ. The second judgement infers the typestates of the fields for a class when it has a declared typestate S. The final judgements are used to state well-formedness properties. The typestate inference rules for expressions can be found in Fig. 6. At any point, there is only one rule that can hold for an expression, as the rules are syntax driven. The ExpVoid, ExpBool, ExpEnum, and ExpNull rules are can be used in any typing context to type the constants. These rules have no effect on the context they type, as noted by the same typing contexts on either side of the expression. Inactive typestate (end) can be removed and added using the ExpStrength and ExpWeak rules. The ExpConst infers a constant type Ω, if for a path o it does not point to a class, and has no effect on the typing context. Mungo enforces linearity, this can be seen in the ExpClass rule, in which the path o can not be this as it would violate linearity at the call site. The RtEquiv states that the output typing context can be Λ'' if it is $\coloneqq$ to Λ' and we know that the expression outputs this context. For the ExpCAsgn rule it first ensures that the expression e has a type Ω which is not a typestate and then assigns this constant to the path o. The ExpTSAsgn rule is similar, except the premise ensures the path o is not this and that the expression e has a typestate as output.

The ExpNew rule can be seen as the exit point of our inference system, it will have built up an inferred type I for path o which is then checked against its declared subtype in the premise of this rule. The output type Ω is void and we notice that the output typing context has the path with an inactive typestate, this is because we can see this as being before the *new* expression, meaning this path has not been initialised yet. The rule has been updated to type the inferred typestate against (v, S) where a helper function *clocks*, shown in the Appendix as Fig. 13, is used to get the set of clocks used in the declared typestate S. For the instance where the declared typestate has no clock constraints, the clocks function will return the empty set which will produce a clock value function with no clock names. This will allow the inferred typestate to type against untimed declared types using the existing rules without rule duplication. The rule also requires that all object fields have an inactive typestate, to ensure these fields have been checked by their ExpNew rule before this object has.

The ExpDelay rule takes as input the typing context Λ' and associates every typestate I with a time constant t. This ensures that time is a global property and all typestates are updated with the passing of time, no matter the context. Classes associated with a *End* typestate are not updated with a time constant, as these classes are inactive so have either finished execution or have yet to be initialised so will have no notion of time.

Rule ExpAction records the method calls of an object associated with a typestate I. The premise ensures a valid method has been called on the path o with the statement $T\ m(T'\ x)\ \{e'\} \in \mathsf{methods}(C)$. The premise has been updated to include the definition of I $\coloneqq$ removeDelays(I', I). The *removeDelays* helper function can be found in the Appendix as Fig. 15. This premise ensures that the receiver method can only be changed by a method expression using *Delay* expressions. It removes these excess expressions before checking the equivalence relation. If the receiver (this) has been changed in any other way, then this premise will fail. The intuition to allow Delay expressions is that a method call should not be able to change the receiver typestate with new method calls or expressions, however time may progress which should be captured by the typestate. The final output typing context, Λ, will have the updated typing of $o : C[\{T\ m(T') : I'\}]$, this attached the method call to this typing while ensures I' which is the typestate with (possibly) updated timing information is added to this path.

The ExpChoice rule infers a typestate for every branch using the input typing context Λ'. These typing contexts are then joined using the *Join* helper function, which is then used as the input typing context for the switch method call itself. The premise of this rule ensures that the return type of the switch method, E is an *enum* type with valid labels.The ExpIf rule is similar to that of a switch rule, but works with only two possible branches.

The ExpLExpr rule is used to infer recursive typestates. It infers the typing context from the expression e and then attaches the recursive label X to every typestate found in the output typing context. The loop label, $\lambda : X$, is given to the input typing context for the expression. This loop label is then used by the ExpContinue rule which replaces inferred typestates with the loop label, X to represent a continued typestate.

3.4 Typestate Inference Examples

The following section is presented to provide examples of the typestate inference. This is to improve ease of understanding and provide worked examples of how our calculus relates to examples shown in Section 4.1. We use the following abbreviations due to space constraints.

- aM = answerMessage

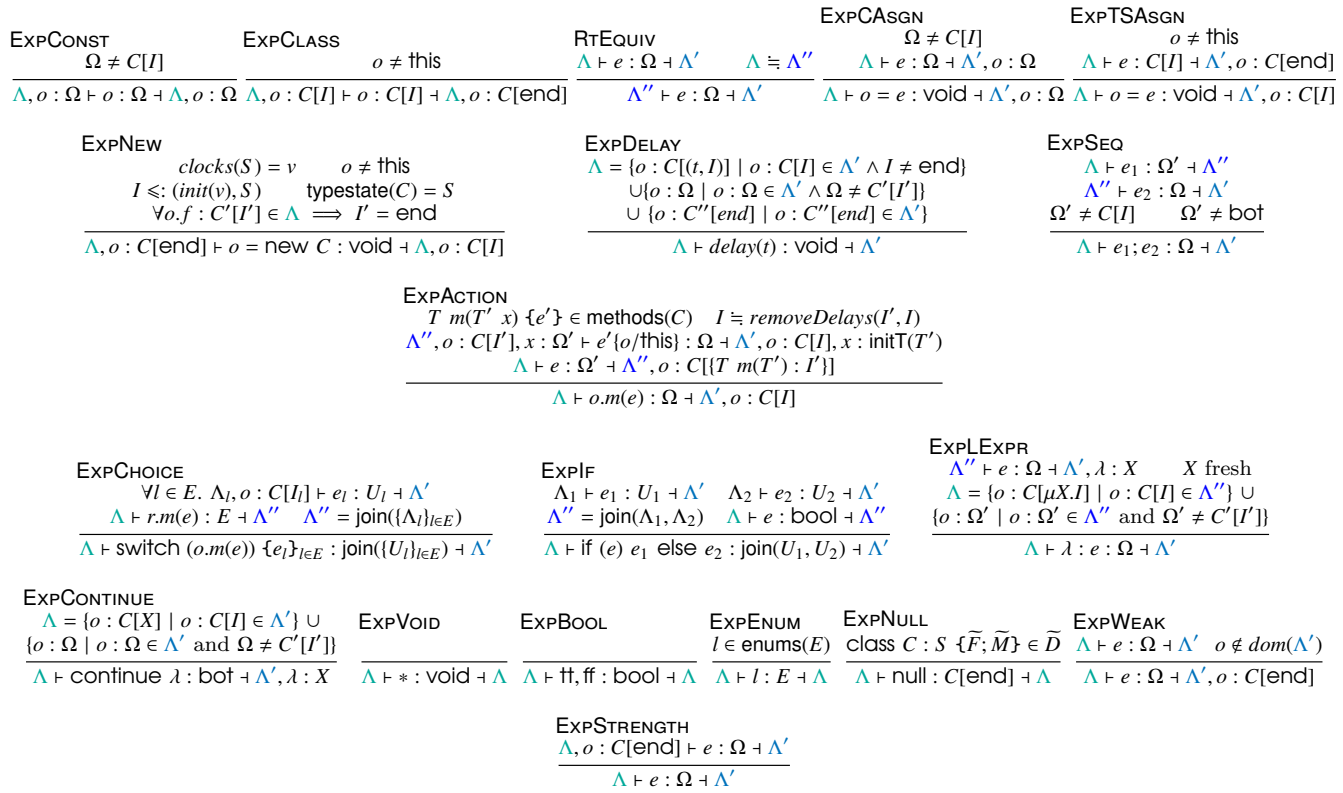

Figure 6: Typestate inference rules for expressions

- MS = MessageStack
- oM = outputMessage
- rS = response:String
- MSU = MessageStackUser
- gR = getResponse
- cFM = checkForMessage

3.4.1 *Simple Delay Rule*

Consider the following code:

```
delay(4); ms.shutdown();
```

Also assume input typing context:
Ω_0 = {ms:MessageStack[end]}
The inference tree for the above code is:

ExpSeq
Delay
$\Lambda = \{r : C[(4, I)] \mid r : C[I] \in \Omega_1\}$
$\Lambda \vdash$ delay(4) : void $\dashv \Omega_1$
ExpAction
(1.1)
$\Omega_1 \vdash$ ms.shutdown() : void $\dashv \Omega_0$
$\Lambda \vdash$ delay(4); ms.shutdown() : void $\dashv \Omega_0$

Where (1.1) is:

ExpAction
void shutdown( ) {} ∈ methods(MessageStack)
end ≒ removeDelays(end, end)
ms:MS[end] ⊢ {r/this} : void ⊣ ms:MessageStack[end]
$\Omega_1 \vdash * : void \dashv$ ms : MessageStack[{void shutdown() : end}]
$\Omega_1 \vdash$ ms.shutdown() : void ⊣ {ms:MessageStack[end]}

with Ω_1 = ms : MessageStack[{void shutdown() : end}]. The final output typing context is Λ = ms: MessageStack[(4, {void shutdown():end})]. The above derivation begins using the ExpSeq rule, which takes the output typing context from the rule on the right as the input for the rule on the left. From the above example, we can highlight the specific premises on the ExpAction rule that are required for this inference. The first premise ensures that the *shutdown()* method is a valid method from the declared *MessageStack* class. Our second premise then ensures that our active receiver is not edited in any way by this method call. The method prefix, *void shutdown()*, is then appended to our typestate before being given as the input typing context to the ExpDelay rule.

We can see from the simple example that 4 is appended to our input typestate I for the path r. In this case, we append the time constant 4 to our output typestate we obtained from our ExpAction rule.

3.4.2 *Delay within a method execution*

Consider the following code fragment

```
String response = msu.getResponse();
ms.answerMessage(response);
```

This fragment of code is used to give an example of when *Delay* expression can occur in an external method call. More complex examples would have delays in separate classes, which can progress time for all typestates.

Assume input typing context:

Ω_0 = response:String, msu:MessageStackUser[l],
ms:MessageStack[l], ms.outMessage:Message[l]

We start with the ExpSeq in which we use ExpAction rule for expression (2.2).

ExpSeq

$$\frac{\Lambda \vdash \text{String response=msu.gR()} : \Omega' \dashv \Omega_1 \quad (2.1) \qquad \Omega_1 \vdash \text{ms.am(response)} : \Omega \dashv \Omega_0 \quad (2.2)}{\Lambda \vdash \text{String response = msu.gR(); ms.aM(response)} : \text{void} \dashv \Omega_0}$$

For expression (2.2) we use ExpAction rule for *void answerMessage(String)*, which in turn makes use of *void set(String)* for our *outMessage* field of our MessageStack object. The output typing context of (2.2) is Ω_1.

Where Ω_1 = response:String, msu:MessageStackUser[l],
ms:MessageStack[{void answerMessage(String):l}],
ms.outMessage:Message[{void set(String):l}]

For expression (2.1) we first use the ExpCAsgn rule in which we assign the result of *getResponse*() to the variable *response*. This rule ensures that the return type of the method call is a constant string by inferring the return type of the method expression. This expression is inferred using the ExpAction rule which is shown below. where (2.1) is:

AsgnC

$$\frac{\text{String} \neq \text{C[l]} \qquad \Lambda \vdash \text{msu.getResponse()} : \text{String} \dashv \Omega_1}{\Lambda \vdash \text{String response = msu.getResponse()} : \text{void} \dashv \Omega_1}$$

The method call, *getResponse* is inferred using the ExpAction below.

Call

$$\frac{\begin{array}{c}\text{String gR( ) \{delay(3); return "42"\}} \in \text{methods(MSU)} \\ \text{I} \coloneqq \text{removeDelays((3,l), l)} \\ \Lambda^4, \text{msu:MSU[(3,l)]} \vdash \text{delay(3);} \\ \text{return "42"} : \text{String} \dashv \Lambda'', \text{msu:MSU[l]} \\ \Lambda \vdash \{\} : \text{void} \dashv \Lambda^4, \text{msu:MSU[\{String gR():(3,l)\}]}\end{array}}{\Lambda \vdash \text{msu.gR()} : \text{String} \dashv \Omega_1, \text{msu:MSU[l]}}$$

Our *getResponse*() method contains a delay expression in its method body. This expression is inferred using ExpSeq with a combination of the ExpDelay and ExpConst rules. This expression body edits *I* by associating the typestate with the time constant 3. Our second premise makes use of the *removeDelays*() function in order to correctly match this time augmented typestate *I* with its equivalent unaugmented typestate. This premise ensures the ExpAction rule correctly accepts this time progression found within the method body.

Our final output typing context, Λ, is as follows.

Λ = response:String,
msu:MSU[{String getResponse():(3,l)}],
ms:MessageStack[(3,{void aM(String):l})],
ms.outMessage:Message[(3,{void set(String):l}])

3.4.3 Branches with different timing constraints

We next give an example of typestate inference for switch branches that demonstrate the switch and recursive rules, as well as how different timing information persists across the branches. Below is an extract from the *MessageStackUser* class that we will use to demonstrate recursive choice.

```
1  loop : {
2    switch(ms.checkForMessage()) {
3      case RECEIVED:
4        Message m = ms.getMessage();
5        String response = msu.
             calculateResponse();
6        ms.answerMessage(response);
7        continue loop;
8      case EMPTY:
9        delay(4);
10       ms.shutdown(); } }
```

Our input typing context Ω_0 is as follows:

Ω_0 = response:String, msu:MessageStackUser[end],
ms:MessageStack[end], ms.outputMessage:[end],
m:Message[end], ms.stackPointer:MyStack[end]

The inference begins by using the ExpLExpr rule, this rule attached the loop variable X to all the typestates that are inferred from the expression e. This allows all typestates to be inferred before attaching the recursive variable to the beginning of these typestates.

ExpLExpr

$$\frac{\begin{array}{c}\Lambda'' \vdash e : \Omega \dashv \Lambda'_0, \text{loop} : X \qquad \text{loop fresh} \\ \Lambda = \{r : C[\mu X.I] \mid r : C[I] \in \Lambda''\} \cup \\ \{r : \Omega' \mid r : \Omega' \in \Lambda'' \text{ and } \Omega' \neq C'[I']\}\end{array}}{\Lambda \vdash \text{loop} : e : \Omega \dashv \Omega_0}$$

Our expression is inferred using the ExpChoice rule. This rule infers each individual enum label branch, before using the *join* helper function to combine these typing contexts.

ExpChoice

$$\frac{\begin{array}{c}\forall l \in \text{CHECK}.\ \Omega_l \vdash e_l : \Omega_l \dashv \Omega_0, \text{loop} : X \\ \Lambda''' = \text{join}(\{\Omega_l\}_{l \in CHECK}) \\ \Lambda'' \vdash \text{ms.cFM()} : \text{CHECK} \dashv \Lambda'''\end{array}}{\Lambda'' \vdash \text{switch (ms.cFM())} \{e_l\}_{l \in \text{CHECK}} : \text{join}(\{\Omega_l\}_{l \in \text{CHECK}}) \dashv \Omega_0, \text{loop:X}}$$

Received branch:

```
1  Message m = ms.getMessage();
2  String response = msu.getResponse();
3  ms.answerMessage(response);
4  continue loop;
```

The RECEIVED branch has four statements which are each individually inferred and combined using the ExpSeq rule which we've seen previously. The inference for line 4 will be given, lines 2 & 3 have already been inferred previously and line 1 will receive this inference.

ExpContinue

$$\frac{\begin{array}{c}\Lambda' = \{r : C[X] \mid r : C[I] \in \Lambda'\} \cup \\ \{r : \Omega \mid r : \Omega \in \Lambda' \text{ and } \Omega \neq C'[I']\}\end{array}}{\Lambda' \vdash \text{continue } loop : \text{bot} \dashv \Omega_0, \text{loop} : X}$$

The ExpContinue rule replaces all typestates found in the input typing context with the recursive variable X which is associated with the *loop* value.

The output typing context, Λ, from the lines 2,3, and 4 is given below. This typing context was inferred from a combination of ExpSeq rules and ExpAction rules as seen previously.

Where Λ = response:String,
msu:MSU[{String getResponse():(3,X)}],
ms:MS[(3, {void aM(String):X})],
ms.outputMessage:[(3,{void Set():X}],
m:Message[X], ms.stackPointer[X]

The final output inference for this branch is given below. It is obtained from using the ExpAction rule and ExpCAsgn on line 1.

Where Ω_R = Λ,
ms:MS[{Message getMessages():
(3,{void answerMessage(String):X})}]

Empty branch:

```
1       delay(4);
2       ms.shutdown();
```

The EMPTY branch involves two expressions, a simple delay followed by the *shutdown*() method. This inference is achieved using the ExpSeq combined with ExpDelay and ExpAction, which we have seen previously. The output typing context for this branch is as follows:

Ω_E = response:String,
msu:MSU[(4, end)],
ms:MessageStack[(4,{void shutdown():end})],
ms.outputMessage:Message[(4,end)], m:Message[(4,end)],
ms.stackPointer:MyStack[(4,end)]

After obtaining the inference for both branches, the ExpChoice rule combines both inferred branches using the *join* helper function. The output of this combination is shown below.

Λ''' = $join(\Omega_R, \Omega_E)$ =
response:String,
msu:MSU:[{String getResponse():(3,X), (4,end)}],
ms:MS[{String gM():(3, {void aM(String):X}),
(4,{void shutdown():end}}],
ms.sP:MyStack[{(4,end),{String pop():X}}],
m:Message[{(3,X), (4, end)}],
ms.oM:Message[{(3, {void set():X}),(4,end)}]

After applying the ExpAction rule to the joined branch in the ExpChoice rule we get the following typing context

Λ'' = Λ''', ms:MS[{Check cFM:{String getMessage():
(3,{void aM():X}),(4,{void shutdown():end})}}]

After getting the output of the switch statement, all typestates are augmented with the recursive label to get the final output typing context as follows:

Λ''' =
response:String,
msu:MSU:[]μX.{String getResponse():(3,X), (4,end)}],
ms:MS[μX.{Check cFM:{String getMessage():
(3,{void aM():X}),(4,{void shutdown():end})}}],
(4,{void shutdown():end}}],
ms.sP:MyStack[μX.{(4,end),{String pop():X}}],
m:Message[μX.{(3,X), (4, end)}],
ms.oM:Message[μX.{(3, {void set():X}),(4,end)}]

3.5 Typechecking Timed Typestates

Once we have inferred typestates for our various classes, we use our **subtyping** relation to ensure the inferred typestates match their declared counterpart following the various (optional) timing constraints. Our subtyping relation takes inspiration from the subtyping relation presented in the original Mungo paper [24].

The subtyping relation is given in Fig. 7. The relation is defined by rules for $I \leqslant: (v, S)$ which is an inferred typestate subtyped against a declared typestate paired with its clock values. Rules StRec1 and StRec2 are used to construct Υ which contains pairs of inferred typestates and declared typestates with its individual clock values. The subtyping algorithm terminates using either the StEnd or StTerm rule. StTerm is used for recursive methods to ensure we don't subtype typestates that have already been compared. The values of the clocks held in v are required to correctly check timed recursive methods. For example, if we had the following constraint

```
[x < 10] void methodName()
```

at one of the final method actions in a recursive block, and each block progressed time by 1 time unit but involved no resets. After 10 loops, our time constraint would be invalid. If we were to change our final time constraint to include a reset including x, then our StTerm would hold and our recursive block would be valid for all possible number of repeats. The StMethod rule matches method prefixes found in both typestates where the declared typestate contains no time constraints. Rule StMethodTime holds when the declared typestate contains a time constraint. The premise states that the inferred type is only subtyped if the prefixes match, and $v \models \delta$ stating that the current clock values held in v are valid under this time constraint. The continuation of this subtyping is then checked on $v' = v[P \rightarrow 0]$ which is the updated values of the clock values after applying the reset predicate. The StRec1 rule unfolds a recursive typestate and adds it to our set of pairs Υ.

External choice actions are checked using the StEnum rule. For every branch held in the inferred branch set $\{B\}$, the premise requires each inferred branch to subtype one of the available enum branches S_l paired with the clock assignment function v'_l. For this subtype relation to hold, the current clock assignment v must hold against the optional clock constraint for this branch δ_l, and where v'_l is the updated clock values based on the reset predicate P_l. The StBranches rule subtypes a set of inferred branches $\{B\}$. Every branch must

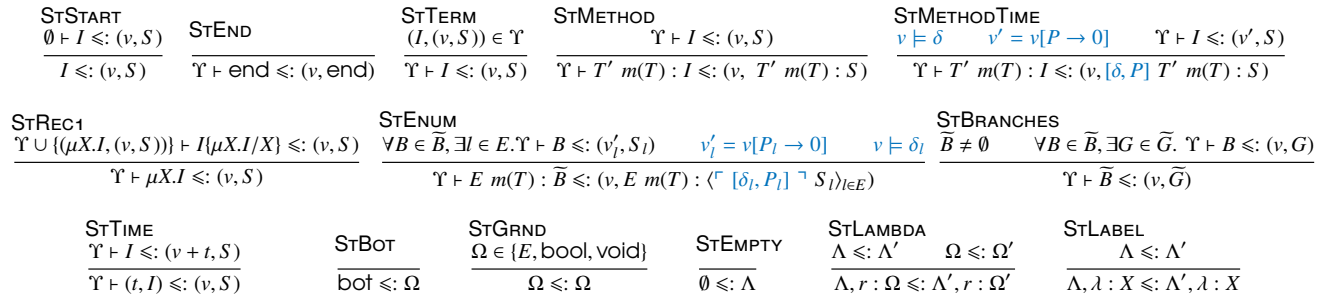

Figure 7: Subtyping relation (symmetric rule StRec2 omitted)

match with an internal choice method signature H. This ensures that each branch subtypes to one of the methods found within a declaration set.

The StTime rule updates the clock assignment function v when (t, I) is subtyped against any declared typestate. This relation holds if the premise where I is a subtype of the updated declared typestate, clock assignment tuple $(v + t, S)$ holds. When reading the subtyping rules from bottom to top, this is equivalent to the number of time constraints found in I reducing as the values are added to the clock assignment function v. This can be seen as time moving 'forward' as we move up the subtyping tree. If read from top to bottom, the terms get larger as the time constant t is added to the left-hand side inferred typestate while the right-hand side stays the same size. This can be seen as moving 'backwards' in time by reducing our clock function v and augmenting the inferred typestate I with t, meaning I occurred after the passing of t time. ExpEnum has been removed from the subtyping of $I \leqslant: (v, S)$ as enums are matched using the StBranches rule as labels are not inferred. As example of how a subtyping example works can be found in Appendix C.

4. IMPLEMENTATION

We present two tools that make up our timed typestate toolchain: Timed Mungo, and Timed StMungo. These tools are extensions of previous work presented by Kouzapas et al [24].

4.1 Timed Mungo

We first present our extension to the Mungo typestate system, Timed Mungo. This extension adds time constraints as presented in Timed Automata [5] to the Typestate protocols that can be attached to Java classes. This extension updates the existing Mungo system, which was created using the JastAdd framework [18]. More details about the implementation is presented in Appendix B. Timed Mungo firstly typechecks using the normal Java type system, and then following the new typestate system. As this is an extension to Mungo, time constraints on method calls are optional and the existing typestate checks are unchanged. As with the original Mungo system, the files are compiled using *javac* and executed in the runtime environment.

A Timed Mungo typestate attaches a state machine, which acts as an object protocol, to a Java class. Each state provides a subset of the defined methods that are valid to be called in that state. Each method call will progress to a new state. The typestate is written using Java like syntax and contains (optional) time constraints on method calls. At compile time, the typestate inference system will infer a typestate, including the time values these methods are called at, and compare it to its declared typestate. They are compared using the subtyping system, which checks if the inferred typestate is a subtype of the declared typestate with a set of clock names. Clock names are defined within the declared typestate file and have their values updated during the subtyping algorithm.

Our extension provides optional time constraints to typestate method calls that are typed checked at compile time. These time constraints are based on individual clocks for each java object. The programmer can write a *tsDelay(x)* expression which will progress time by a time unit x. Time constraints also contain a reset predicate, which allows recursive protocols to be defined.

Example. To introduce the Timed Mungo extension, we present an example of a Message Stack data structure and its associated timed typestate definition. Following the given enumerated type:

```
enum Check { RECEIVED, EMPTY }
```

then one possible timed typestate protocol for a message stack is as follows:

```
1 typestate MessageStackProtocol {
2   Stack     = {Check checkForMessage(): <
        RECEIVED: [xm<10, {xm}] Present,
        EMPTY: [xm<10 EndState>}
3   Received = {[xm < 10, {xm}] void
        answerMessage(String) : Stack, [xm <
         5] void ignoreMessage() : Stack}
4   EndState  = {void shutDownStack() : end}
5   Present = {[xm < 2] Message getMessage()
        : Received}
```

The definition specifies that the initial state is Stack. In this state the checkForMessage() is defined which queries the MessageStack to check if a message has been sent and will progress to a different state based on the returned enum. The enum is returned by the separate Stack class and models an external choice operator from Session Type theory. Given a RECEIVED choice, the typestate progresses to the Present state. If EMPTY is returned, it will progress to the EndState which terminates the stack. This method also presents two separate time constraints. The first stating that the RECEIVED choice can only be received if the xm clock has a value less than 10 time units. Once this clock constraint is checked, it will reset the value held within this

clock. This reset is required for the future time constraints and the ability to continuously looping through this state, checking all the messages on the stack. The EMPTY branch has a similar clock constraint, but does not reset the value of the clock. In the Present state, the user must call the getMessage() function within 2 time units. This will pop the message and progress to the Received state. In this state, the user can answer the message within 10 time units or ignore the message, both messages return to the initial Stack state.

A MessageStack implementation can now be defined which has the MessageStackProtcol attached using the **@Typestate("MessageStackProtocol")** annotation.

```
1  @Typestate("MessageStackProtocol")
2  public class MessageStack{
3      private MyStack stackPointer;
4      Message outputMessage;
5      MessageStack(MyStack stackPointer,
           Message messagePointer) {
6          this.stackPointer = stackPointer;
7          this.outputMessage =
               messagePointer; }
8      Check checkForMessage(){
9          return this.stackPointer.status; }
10     void answerMessage(String response){
11         this.outputMessage.setMessage(
               response); }
12     Message getMessage(){
13         return new Message(this.
               stackPointer.pop()); }
14     void ignoreMessage(){
15         this.outputMessage.setMessage("
               Message ignored"); }
16     void shutdownStack(){}
17 }
```

A MessageStack client is then defined which uses an instance of the MessageStack class. Timed Mungo will verify that methods are called in the correct order and that they are valid against their time constraints.

```
1  class MessageStackUser{
2    public String getResponse(){
3      tsDelay(3);
4      return "42"; }
5    public MessageStackUser(){}
6    public static void main(String[] args){
7      MyStack stack = new MyStack();
8      MessageFiller.fillMessages(stack);
9      Message messagePointer = new Message
           ("");
10     MessageStack ms = new MessageStack(
           stack, messagePointer);
11     MessageStackUser msu = new
           MessageStackUser();
12     loop : do{
13       switch(ms.checkForMessage()){
14         case RECEIVED:
15           Message m = ms.getMessage();
16           System.out.println(m.toString()
                 ;
17           String response = msu.
                 getResponse();
18           ms.answerMessage(response);
19           continue loop;
20         case EMPTY:
21           tsDelay(4);
22           ms.shutdownStack();
23           break loop; }
24     } while(true); } }
```

Three helper classes are defined for the client. MyStack is a simple class that has a private stack field that uses a standard Java stack as well as a status field which will return the required Check enum for the internal choice. This status is updated within the push and pop methods of the class. Checking if there is a method that should be returned.

Message is a simple class containing a private String field and the ability to set and return this String.

MessageFiller takes a MyStack classes and pushes messages onto the stack, this is to simulate an external class that has access to this stack pointer and updates the stack with *Messages* from different clients.

Time Progression The tsDelay(x) expression as seen in the EMPTY case progresses the individual clocks of all typestates forward by a time unit x. This expression as presented by Timed Multiparty Session Types [9] models a computationally intense operation which would take a certain time frame to perform. This expression has no side effects and is only used by the typing system. If a typestate definition has no time constraints, then delay expressions will have no effect on the validity of classes that conform to this typestate. We can see that the tsDelay(3) after the EMPTY case that the clock xm will progress by 3 time units. This models some computational time required before shutting down the MessageStack.

Time progression within other methods The getResponse() method illustrates the feature of delay methods held within external method calls not explicitly called by the typestate class, in this case the MessageStack. This allows Timed Mungo to check instances where the user has added tsDelays in external functions or API calls that will take a certain time unit without needing to add tsDelays in the location the type checked class is being used. In this example, we are modelling a situation where it takes 3 time units to generate an appropriate response. If we were to execute this method three times, we continue to progress the xm clock until it would no longer hold for the answerMessage() time constraint as the clock would be greater than 10, invalidating this method call.

Recursion and timing in different branches The loop do-while blocks illustrates an example of a recursive loop that consumes the external choice held within the MyStack class. At each iteration, the status of the stack is queried, and the relevant branch is chosen. If RECEIVED where to be consumed, then time will pass 5 time units before looping. As the time constraint held by the answerMessage() method contains a reset predicate of {xm} then the clock values are reset to zero after each iteration. This allows the same clock constraint on each method call to be valid at each iteration. Each branch of the switch statement can progress at different times. In this example, the EMPTY branch will finish with time value 4, while the RECEIVE branch can loop multiple times and progress time by 3 before being reset.

4.2 Timed StMungo

The original StMungo [24] tool translates local protocols written in the Scribble [36] protocol language into Mungo typestate specifications, while also providing initial skeleton implementation files. The translated protocol can then be used in Mungo typechecking.

A user can write a global protocol in Scribble which is checked for well-formedness before being projected into local protocols following the theory of *Global* and *Local* session

types as presented in Multiparty Session Types [21]. The global session type describes all possible communications between different parties and interleaving between them. A local session type is a projection to a single party in this global protocol, it only depicts communication methods from a single view point.

An extension to Scribble was presented by Neykova et al. [29] which updated Scribble definitions with clocks, clock constraints, and resets following definitions found in Timed Multiparty Session Types [9]. Scribble can now define *Timed Global Protocols* which define the time constraints of interactions between multiparty parties, as well as existing constraints on causality and data types. Their extension performs a consistency check on global protocols, which ensures two properties hold; *wait-freedom* and *feasibility*. These conditions are required to ensure well-formed protocols and are assumptions required by our extended theory. After performing the consistency check, Scribble projects a protocol into the individual *timed local protocols*.

Our extension to StMungo, **Timed StMungo**, updates the existing system to translate timed local protocols into their timed typestate specifications. This allows users to design and write timed global specifications in Scribble before translating and type checking Java classes corresponding to their timed typestate definitions. These timed typestate protocols allow users to implement timed communication protocols which guarantee communication safety and soundness, as seen in Section 5.2.

Timed StMungo is described using a timed multiparty protocol that models a book search example. The following protocol takes inspiration from the log crawling example presented by Neykova et al. [29]. This protocol has three parties: User (U), who is searching for a book to read; Server S, that returns possible book options to the user; and Library L, a personal user library that makes a record of the books that the user has taken out. After the user asks the server for a book, they can choose to add this book to their personal library or skip it. Then can then choose to search for more books, which recursively loops the protocol, or end the connection. The timed global protocol is defined as below.

```
global protocol BookSearch(role U, role S,
    role L) {
[@U:xu<1, reset(xu)][@S:xs=1, reset(xs)]
books(String) from U to S;
rec Loop{
  [@S:xs<10][@U:xu>10 & xu<12]
  result(Book) from S to U;
  choice at U{
    [@U:xu=12][@L:xl>12, reset(xl)]
    addBook(Book) from U to L;
  } or {
    [@U:xu=12][@L:xl>12, reset(xl)]
    skipBook() from U to L; }
  choice at U{
    [@U:xu=13][@S:xs=14,reset(xs)]
    moreBooks(String) from U to S;
    [@U:xu=13, reset(xu)][@L:xl>=14, reset
        (ul)]
    moreBooks(String) from U to L;
    continue Loop;
  } or {
    [@U:xu=13][@S:xs=14]
    end() from U to S;
    [@U:xu=13][@L:xl>=14]
    end() from U to L; } } }
```

The above protocol lists time constraints on the various actions between the parties. In order to ensure the server doesn't try to read a message from S before it is sent, they must wait for time to pass until xs = 1, as U must send this request before their clock equals 1, this read will occur at the correct timing. This condition on U, xu < 1, is required to make this action **wait-free**, which is a required property for our implementation and theory. Reset predicates are shown in the constraints of the moreBooks(String) from U to L. This reset predicate is required for the time constraints to be well-formed for the next iteration of the loop. Another property required by this implementation of Scribble is **feasibility**, which states that "A protocol will not get stuck due to some unsatisfiable constraint" [29]. If we were to change the time constraint for result(Book) to `[@S:xs<10][@U:xu<5]` then we would have an unsatisfiable condition, as if S sends the message at xs=9 then the constraint would not be satisfiable for U.

Scribble would check the above protocol for well-formedness and then create a local protocol for each role, known as *endpoint projection* [21]. Below we show the local protocol for the Library.

```
local protocol BookSearch_L(role U, role S
    , self L) {
rec Loop{
  choice at U{
    [@L:xl>12, reset(xl)]
    addBook(Book) from U;
  } or {
    [@L:xl>12, reset(xl)]
    skipBook() from U; }
  choice at U{
    [@L:xl>=14, reset(xl)]
    moreBooks(String) from U;
    continue Loop;
  } or {
    [@L:xl>=14]
    end() from U; } } }
```

Observe that for this local protocol no information about messages sent from U to S are included. This follows the intuition that local protocols can be converted to single Java classes as information about other roles is not required, this includes time constraints. Timed StMungo converts this local protocol into the following timed typestate specification:

```
typestate LProtocol {
  State0 = {
    LChoice1 receive_LChoice1LabelFromU()
        :<ADDBOOK: State1, SKIPBOOK:
        State2> }
  State1 = {
    [xl>12, {xl}] null
        receive_addBookBookFromU(): State3
         }
  State2 = {
    [xl>12, {xl}] void
        receive_skipBookFromU(): State3 }
  State3 = {
    LChoice2 receive_LChoice2LabelFromU():
         <MOREBOOKS: State4, END: State5>
        }
  State4 = {
    [xl>=14, {xl}] null
        receive_moreBooksStringFromU():
        State0 }
  State5 = {
    [xl>=14] void receive_endFromU(): end
        } }
```

5. EVALUATION

In this section, we provide evaluation details based around an extended Timed SMTP case study which we use to evaluate our two extended tools. We also provide details about properties we can guarantee for our system: **progress** and **subject reduction**.

5.1 Evaluation of Timed SMTP

In the original Mungo paper [24] an SMTP case study was carried out to check how practical their toolchain was. They developed an SMTP client, which they then typechecked using Mungo. *Simple Mail Transfer Protocol* (SMTP) is a TCP based mail transfer protocol. SMTP involves a client and server exchanging text-based commands, such as "MAIL" which allows the client to describe the address of the sender. If we consider the version of SMTP presented in RFC 5321 [23] we can observe the notion of timeouts on the server and client side for various communications. For example, we have "An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender". In order to evaluate the Timed Mungo toolchain, the case study was updated to capture these timeout constraints. Below is an example of the updated SMTP client class.

```
local protocol SMTP_C(role S, self C) {
[@C: xc = 300, reset(xc)]
_220(String) from S;
choice at C{
  ...
  [@C: xc < 300, reset(xc)]
  mail(String) to S;
  choice at S{ ...
  } or {
    data(String) to S;
    [@C: xc = 120, reset(xc)]
    _354(String) from S;
    rec Z3 {
    ...
    } or {
      atad(String) to S;
      [@C: xc = 600, reset(xc)]
      _250(String) from S;
          continue Z1; } }
```

Below is the corresponding server process.

```
local protocol SMTP_C(self S, role C) {
[@S: xs < 300, reset(xs)]
_220(String) to C;
choice at C{
  ...
  [@S: xs = 300, reset(xs)]
  mail(String) from C;
  choice at S{ ...
  } or {
    data(String) to C;
    [@S: xs < 120, reset(xs)]
    _354(String) from C;
    rec Z3 { ...
    } or {
      atad(String) to C;
      [@S: xs < 600, reset(xs)]
      _250(String) from C;
          continue Z1; } }
```

Our updated protocol definitions capture the SMTP timeouts discussed in RFC 5321. Including a 5-minute timeout for the client to receive the initial 220 greeting message. This captured by the clock constrains held in both protocols by 220(String) from S and 220(String) to C. The clock constraint on the server process is

```
[@S: xs < 300, reset(xs)]
```

This means the server must send the greeting message within 5 minutes. The client will then check for this message when their clock equals 5 minutes. As time is global, it will ensure this has no overlap and this process is wait-free and feasible. After translating these local protocols using Timed StMungo, the typestates can be attached to the generated API classes. The classes are then updated with relevant *delay* expressions to validate the existing time constraints. These timeouts can be captured by the class usage, as shown by the below code snippet.

```
CRole currentC =  new CRole();
...
tsDelay(300);
SMTPMessage payload1 = SMTPMessage.Parse(
    currentC.receive_220StringFromS());
```

The above code snippet displays the use of a delay expression to model a passing of time that would make the next receive_220StringFromS() method call valid. This example currently models a client waiting until the end of its timeout before it queries the message from S.

Evaluation Our time constraint theory is based upon the work presented by Bocchi et al. [9]. This was chosen as they present a timed multiparty session type (TMST) theory which was used to extend the existing Mungo theory based on untimed multiparty session types. Other state of the arts work were explored but were missing the required multiparty features required by Mungo. The update to Scribble presented in Timed runtime monitoring [29] is also based upon TMSP, this therefore would ensure our update to StMungo follows the state of the art Scribble protocol language.

As our time extension work is based upon TMST, certain properties had to be captured. Including *wait-freedom* and *feasibility*. The above SMTP timeouts need to follow the need for *wait-freedom*. This means the sender has until the receivers' timeout to send a message, and the receiver must access this message at the timeout. This is not a perfect modelling of these SMTP timeouts as the timeouts themselves are inherently non-wait-free as previously discussed by Bocchi et al. [8]. Our existing work still allows (synchronous) wait-free time constraints to be captured, but it is relevant to note that future work will concern research into capturing asynchronous time constraints for typestates.

5.2 Typestate Inference Properties

After creating our formalism of timed typestates we constructed proofs for progress and type preservation following existing properties found in the original Mungo formalisation [24]. The proof is given in Appendix A.

THEOREM 1. *(*Progress and Type Preservation*) Assuming a program context $\widetilde{D}$, let e be a run time expression and suppose $\Lambda \vdash h, e : U \dashv \Lambda''$. Then the following two statements hold:*

- *Either e is a value, or there exist unique ℓ, h' and e' such that $h, e \xrightarrow{\ell} h', e'$, and there exist Λ' and U' such that $\Lambda \xrightarrow{\ell} \Lambda'$ and $\Lambda' \vdash h', e' : U' \dashv \Lambda''$ and $U' \leqslant: U$.*
- *For any $r : C[I] \in \Lambda$ and $r : C[I'] \in \Lambda'$, if there exists*

(v, S) *and* s *such that* $I \leqslant: (v, S)$ *and* $(v, S) \xrightarrow{s} (v', S')$, *then we have*

- *For any time constraint* δ *in the form of* S, $v \vDash \delta$.
- *For any clocks* $\{\pi | \pi \in v \cap v'\}$ *either* $\exists \pi. v'(\pi) = 0$ *or* $\forall \pi. v'(\pi) \geq v(\pi)$.
- $I' \leqslant: (v', S')$

The type preservation property implies runtime and time safety for every object, where the sequence of method calls is a path within the declared typestate related to its clock values. If we assume an action, l, is a method call then by the above property we know it is a step in the inferred path. We can use this reasoning for a path of method calls and delay expressions within an inferred typestate, I. From our inference rules in Section 3.3, we know object creation is carried out using the ExpNew rule. In its premise, it states that the inferred typestate subtypes the declared subtype paired with its clock values. Therefore, we can imply runtime and time safety.

The following two corollaries can be inferred from the above theorem:

Corollary 1 (*Protocol termination*) If an expression is well typed against its declared subtype then it will continue to progress until it reaches an end termination state.

More formally, if we have that $\Lambda \vdash h, e : U \dashv \Lambda''$ then $\forall r : C[I] \in \Lambda$ we can find a sequence of actions $\bar{\ell}$ such that $r : C[end] \in \Lambda'''$ where $e \xrightarrow{\bar{\ell}} e'$ and $\Lambda'' \xrightarrow{\bar{\ell}} \Lambda'''$.

Corollary 2 (*Time Progression*) Time in the typing system will only ever positively increase or stay the same after expression calls.

More formally, if we have $I \leqslant: (v, S)$, we can find a clock $x \in v$ such that for a sequence of actions $\bar{\ell}$ such that $(v, S) \xrightarrow{\bar{\ell}} (v', S')$ we have $v(x) <= v'(x)$.

The above proof implies *time safety*, but does not explicitly prove this. Future work will be carried out to provide proofs in this area, but we are confident the system implies time safety due to the nature of the time theory adapted for typestates. As we choose to adapt the *wait-freedom* and *feasibility* conditions from Timed Multiparty Session types for our time system, we are confident our above proof is sufficient for safe usage. However, it is important to note that a prove has not been provided and is left as future work.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented extensions to Mungo and StMungo in order to capture time constraints as presented in Timed Multiparty Session Types [9]. This extends the static type checking capabilities of Mungo to capture and verify time constraints following our extended timed typestate theory. The Timed Mungo system extends Java typestate definitions with time constrains, clocks, and delay expressions. Our Timed StMungo tool can translate timed Scribble local protocols into timed typestate protocols, providing a timed typestate toolchain to allow programmers to develop timed protocols easily. We presented our updated timed typestate theory, which defines a timed typestate inference system. Progress and Subject Reduction were proved for our updated theory, implying runtime and time safety for every object.

Future Work Our current system requires time constraints to be *wait-free*, this leads to some time constraints to be modelled differently for our tool, such as the timeout constraints found within SMTP. Future work would involve updating our tool to capture the time theory presented in Asynchronous Timed Communication [8] which allows non wait-free protocols to be captured in the binary setting. Work can be carried out to provide a separation or new tool that can be used to capture these timed binary protocols while ensuring processes conform to their time constraints.

Currently, our formalisation does not include a formal time safety proof. This would be required for reliable static type checking of time constraints and would be an important next step for this work.

Programmers using our Timed Mungo tool can use *tsDelay* to capture the progression of time for a class. While this allows multiple different functions and computations to be modelled, it can be unnatural for a programmer to carry out accurately. Future work can be carried out to research new ways to capture time progression statically for typestates that reduces the need for programmers to progress time themselves.

Acknowledgments. This project would not have been possible without the wonderful support from my supervisor Dr Ornela Dardha. I would like to thank her excellent feedback and guidance throughout a challenging and interesting dissertation.

7. REFERENCES

[1] Beaver - a LALR Parser Generator.

[2] ExtendJ - The JastAdd Extensible Java Compiler.

[3] S. Akshay, P. Gastin, M. Mukund, and K. N. Kumar. Model checking time-constrained scenario-based specifications. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, Oct. 2009. Association for Computing Machinery.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.

[6] M. Bartoletti, T. Cimoli, and M. Murgia. Timed Session Types. *Logical Methods in Computer Science ; Volume 13*, page Issue 4 ; 18605974, 2017. Medium: PDF Publisher: Episciences.org.

[7] L. Bocchi, J. Lange, and N. Yoshida. Meeting Deadlines Together. page 14 pages, 2015. Artwork Size: 14 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[8] L. Bocchi, M. Murgia, V. T. Vasconcelos, and N. Yoshida. Asynchronous Timed Session Types: From Duality to Time-Sensitive Processes. In *Programming Languages and Systems*, volume 11423. Springer International Publishing, Cham.

[9] L. Bocchi, W. Yang, and N. Yoshida. Timed Multiparty Session Types. In P. Baldan and D. Gorla, editors, *CONCUR 2014 – Concurrency Theory*, volume 8704, pages 419–434. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Series Title: Lecture Notes in Computer Science.

[10] M. Bravetti, A. Francalanza, I. Golovanov, H. Hüttel, M. S. Jakobsen, M. K. Kettunen, and A. Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Programming Languages and Systems*, Lecture Notes in Computer Science, Cham. Springer International Publishing.

[11] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science*, 410(2-3):142–167, Feb. 2009.

[12] S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 917–934, New York, NY, USA, Oct. 2015. Association for Computing Machinery.

[13] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On Progress for Structured Communications. In *Trustworthy Global Computing*, Lecture Notes in Computer Science, Berlin, Heidelberg. Springer.

[14] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, and N. Yoshida. Bounded Session Types for Object Oriented Languages. In *Formal Methods for Components and Objects*, volume 4709. Springer Berlin Heidelberg, Berlin, Heidelberg.

[15] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP 2006 – Object-Oriented Programming*, volume 4067. Springer Berlin Heidelberg, Berlin, Heidelberg.

[16] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object-Oriented Language with Session Types. In *Trustworthy Global Computing*, volume 3705. Springer Berlin Heidelberg, Berlin, Heidelberg.

[17] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular Session Types for Distributed Object-Oriented Programming. page 25.

[18] G. Hedin. *An Introductory Tutorial on JastAdd Attribute Grammars*. Lecture Notes in Computer Science. Springer.

[19] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715. Springer Berlin Heidelberg, Berlin, Heidelberg.

[20] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, volume 1381. Springer Berlin Heidelberg, Berlin, Heidelberg.

[21] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 273–284, New York, NY, USA, Jan. 2008. Association for Computing Machinery.

[22] M. Jakobsen, A. Ravier, and O. Dardha. Papaya: Global Typestate Analysis of Aliased Objects. In *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021, pages 1–13, New York, NY, USA, Sept. 2021. Association for Computing Machinery.

[23] D. J. C. Klensin. Simple Mail Transfer Protocol. RFC 5321, Oct. 2008.

[24] D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming*, 155:52–75, Apr. 2018.

[25] P. Krcal and W. Yi. Communicating Timed Automata: The More Synchronous, the More Difficult to Verify. In *Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Heidelberg. Springer.

[26] H. A. López and J. A. Pérez. Time and Exceptional Behavior in Multiparty Structured Interactions. In *Web Services and Formal Methods*, volume 7176. Springer Berlin Heidelberg, Berlin, Heidelberg.

[27] A. Miu, F. Ferreira, N. Yoshida, and F. Zhou. Communication-safe web programming in TypeScript with routed multiparty session types. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, pages 94–106, New York, NY, USA, Mar. 2021. Association for Computing Machinery.

[28] J. D. d. L. Mota. Coping with the reality: adding crucial features to a typestate-oriented language. Feb. 2021. Accepted: 2021-09-29T10:28:46Z.

[29] R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, Sept. 2017.

[30] L. Padovani. Deadlock-Free Typestate-Oriented Programming. Dec. 2017.

[31] N. Saeedloei and G. Gupta. Timed $$\pi $$-Calculus. In *Trustworthy Global Computing*, Lecture Notes in Computer Science, Cham. Springer International Publishing.

[32] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, Jan. 1986. Conference Name: IEEE Transactions on Software Engineering.

[33] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and Tanter. First-class state change in plaid. *ACM SIGPLAN Notices*, 46(10):713–732, Oct. 2011.

[34] V. T. Vasconcelos et al. Sessions, from types to programming languages. *Bull. EATCS*, 103:53–73, 2011.

[35] H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In *Programming Languages and Systems*, Lecture Notes in Computer Science. Springer.

[36] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble Protocol Language. In *Trustworthy Global Computing*, Lecture Notes in Computer Science, Cham. Springer International Publishing.

APPENDIX

A. PROGRESS AND SUBJECT REDUCTION

The following proof is an extension and update of the proof presented in the Original Mungo paper [24]. The main updates involving ensuring the new calculus still has the progress and subject reduction properties, while also ensuring time constraints work as expected.

In order to ease understanding of the following proof, previous definitions have been repeated here for clarity. We model time using the theory presented in Timed Automata [5]. Let Π be a set of clocks ranging over $\pi_1, \pi_2, ..., \pi_n$, taking values in $\mathbb{R}_{\geq 0}$. $v : \Pi \to \mathbb{R}_{\geq 0}$, is a *clock assignment* which returns the times of clocks in Π. We write $v + t$ assignment mapping all $\pi \in \Pi$ to $v(\pi)+t$. The initial assignment mapping all clocks to 0 is written as v_0. t is a time constant in $\mathbb{Q}_{\geq 0}$. We write $v \models \delta$ when δ is satisfied by v. A reset predicate P contains a subset of Π. When P is $\varnothing$ no resets occur and clocks are untouched, else for each $\pi \in \Pi$, π is set to 0. We write $v[P \to 0]$ for the reset of all clocks in P to 0. Inferred typestates, I, are subtyped against (v,S), a declared subtype with a clock assignment function that keeps track of clock assignments throughout the subtyping relation.

We present the Theorem below and give the proof after providing relevant auxiliary definitions, and semantics.

Theorem 1: Assuming a program context $\widetilde{D}$, let e be a run time expression and suppose $\Lambda \vdash h, e : \Omega \dashv \Lambda''$. Then the following two statements hold:

- Either e is a value, or there exist unique ℓ, h' and e' such that $h, e \xrightarrow{\ell} h', e'$, and there exist Λ' and Ω' such that $\Lambda \xrightarrow{\ell} \Lambda'$ and $\Lambda' \vdash h', e' : \Omega' \dashv \Lambda''$ and $\Omega' \leqslant: \Omega$.
- For any $r : C[I] \in \Lambda$ and $r : C[I'] \in \Lambda'$, if there exists (v, S) and s such that $I \leqslant: (v, S)$ and $(v, S) \xrightarrow{s} (v', S')$, then we have
  - For any time constraint δ in the form of S, $v \models \delta$.
  - For any clocks $\{\pi | \pi \in v \cap v'\}$ either $\exists \pi . v'(\pi) = 0$ or $\forall \pi . v'(\pi) \geq v(\pi)$.
  - $I' \leqslant: (v', S')$

A.1 Required Lemmas

The following required auxiliary results are from Mungo [24] with only slight updates to match new syntax. We use $\Lambda\{v : \Omega\}$ to denote Λ where the value v is updated to the type Ω. Proofs for these auxiliary results can be found in the original Mungo paper.

Lemma 4 - Typability of Heap Update
Let h be a heap and o a runtime path such that $\Lambda \vdash h$ and $o : \Omega \in \Lambda$.
1. If $\Omega \neq C[I]$ and $\Lambda \vdash a : \Omega \dashv \Lambda$, then $\Lambda \vdash h\{o \mapsto a\}$.
2. If $\Omega = C[I]$ and $\Lambda' \vdash a : C[I'] \dashv \Lambda'$, then $\Lambda' \vdash h\{o \mapsto a\}$, for $\Lambda' = \Lambda\{o : C[I']\}$.

Lemma 5 - Replacement
If
- d is a derivation for $\Lambda \vdash \mathcal{E}[e] : \Omega \dashv \Lambda''$,
- d' is a subderivation of d concluding $\Lambda \vdash e : \Omega_e \dashv \Lambda_e$,
- the position of d' in d corresponds to the position of the hole in $\mathcal{E}$,
- $\Lambda' \vdash e' : \Omega_{e'} \dashv \Lambda_e$, such that $\Omega_{e'} \leqslant_{\text{sbt}} \Omega_e$,

then $\Lambda' \vdash \mathcal{E}[e'] : \Omega' \dashv \Lambda''$ such that $\Omega' \leqslant_{\text{sbt}} \Omega$.

Lemma 6 - Substitution
1. If $\Lambda, x : \Omega' \vdash e : \Omega \dashv \Lambda'$ and $\Lambda\{v : \Omega'\} \vdash v : \Omega' \dashv \Lambda''$, then $\Lambda\{v : \Omega'\} \vdash e\{v/x\} : \Omega \dashv \Lambda'$.
2. Assume $\Lambda_1 \vdash e : \Omega \dashv \Lambda', \lambda : X$ and $\Lambda_2 \vdash e' : \Omega \dashv \Lambda'$. Then, $\Lambda \vdash e\{e'/\mathsf{continue}\ \lambda\} : \Omega \dashv \Lambda'$ with

$$\begin{aligned}\Lambda \quad = \quad & \{o : C[I\{I'/X\}] \mid o : C[I] \in \Lambda_1 \text{ and } o : C[I'] \in \Lambda_2\} \\ \cup \quad & \{o : \Omega' \mid o : \Omega' \in (\Lambda_1 \cup \Lambda_2)\Omega' \neq C'[I']\} \cup \Lambda_1 \backslash \Lambda_2 \cup \Lambda_2 \backslash \Lambda_1\end{aligned}$$

Lemma 7 - Subtyping and Join
The following relate subtyping and join on inferred types Ω and typing contexts Λ.
1. Let Ω, Ω' be inferred types such that $\text{join}(\Omega, \Omega')$ is defined. Then, $\Omega \leqslant_{\text{sbt}} \text{join}(\Omega, \Omega')$ and $\Omega' \leqslant_{\text{sbt}} \text{join}(\Omega, \Omega')$.
2. Let Λ, Λ' be such that $\text{join}(\Lambda, \Lambda')$ is defined. Then, $\Lambda \leqslant_{\text{sbt}} \text{join}(\Lambda, \Lambda')$ and $\Lambda' \leqslant_{\text{sbt}} \text{join}(\Lambda, \Lambda')$.

Lemma 8 - Typability of Subterms
If d is a derivation for $\Lambda \vdash \mathcal{E}[e] : \Omega \dashv \Lambda''$ then there exist Λ' and Ω' such that d has a subderivation d' concluding $\Lambda \vdash e : \Omega' \dashv \Lambda'$ and the position of d' in d corresponds to the position of the hole in $\mathcal{E}$.

$$\begin{array}{rll}
o & ::= & C[\widetilde{f : o}] \mid c & \textbf{Heap value} \\
r & ::= & \mathsf{root} \mid r.f & \textbf{Runtime path} \\
e & ::= & \ldots \mid e@r & \textbf{Expression} \\
v & ::= & c \mid r & \textbf{Value} \\
S & ::= & \ldots \mid \langle \ulcorner \kappa_l \urcorner\ S_l \rangle_{l \in E} & \textbf{Typestate} \\
s & ::= & T\ m(T) \mid E\ m(T) : l \mid l \mid \mathit{delay}(t) & \textbf{Typestate action} \\
I & ::= & \ldots \mid \langle I_l \rangle_{l \in E} & \textbf{Inferred-Typestate} \\
\mathcal{E} & ::= & [] \mid r.m(\mathcal{E}) \mid r.f = \mathcal{E} \mid \mathcal{E}; e & \\
 & & \mid \mathsf{switch}\ (\mathcal{E})\ \{e_l\}_{l \in E} \mid \mathcal{E}@r & \\
 & & \mid \mathsf{if}\ (\mathcal{E})\ e\ \mathsf{else}\ e & \textbf{Context} \\
\ell & ::= & r.f.\mathsf{new}\ C \mid r.\langle l \rangle \mid r.T\ m\ T' \mid r.f = v & \\
 & & \mid \tau \mid \mathsf{if} \mid \mathit{delay}(t) & \textbf{Action}
\end{array}$$

Figure 8: Runtime syntax

A.2 Semantics

In this section we present the updated semantics of our formalisation which are required for the *Progress* and *Subject Reduction* proofs. These semantics are extensions of the semantics presented in Mungo [24].

A.2.1 Configurations and runtime syntax.

Fig. 8 shows the runtime syntax of Timed Mungo. This runtime syntax extends that found in the original Mungo. The original syntax will briefly be covered. h, e is a *configuration* which contains a *heap* h and *runtime expression* e. The heap stores objects represented by the class C and it's fields $\{f : a\}$ where a is either a constant or an object. The type system enforces linearity, so the heap has no cycles or sharing.

A *runtime expression* e where paths are replaced with *runtime paths* which refer to heap values. These paths are

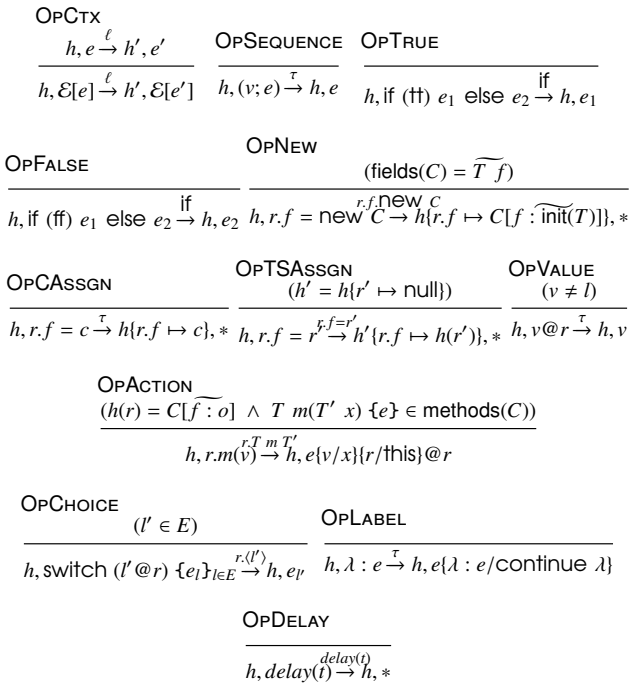

Figure 9: Operational semantics

either root or a composite path. Expressions are tagged with o so the active receiver can be tracked. Their expressions are either values, which are constants or runtime paths, or *contexts*. Contexts are standard expressions with a hole. Runtime typestate labels have been updated to include the optional time constraints. This allows individual paths to have independent time constraints. The Inferred Typestates I have also been updated to include the set of labels.

An action ℓ is denoted by a label used to annotate the operational semantics. These actions include object creation, enum value choice, method calls, field assignment, conditional labels, and silent labels. A new *delay(t)* action has been extended to this syntax to represent the passage of time.

A.2.2 Operational Semantics

Heap access and update functions that are used by the reduction relation in Fig. 9 are defined below:

$$\left.\begin{array}{lcl} h(\mathsf{root}) & = & h \\ h(o.f) & = & a \\ h\{o.f \mapsto a'\} & = & h\{o \mapsto C[\widetilde{f : a}, f : a']\} \end{array}\right\} \text{ if } \quad h(o) = C[\widetilde{f : a}, f : a]$$

The operational semantics are defined in Fig. 9, the semantics extend those defined by Mungo with a new relation declared for the delay expressions. Rule OpCtx uses an evaluation context to lift the semantic rules of an arbitrary expression. Rule OpSequence discards a value v using the silent action to continue with the expression e. Rules OpTrue and OpFalse are standard if-else rules. Rule OpNew which follows an action labelled by o.f.new C and overwrites the contents of the $o.f$ field. This is defined using the *init*() function described below. Rule OpCAssgn stores a constant into the path, as it has no typestate this rule can be used with no restrictions. Mungo enforces linearity as seen by Rule R-AsgnR which moves the object located at o' to $o.f$ leaving null in the previous location.

Rule OpAction is labelled by a method $o.T\ m\ T'$, it first finds the object located on the heap by o, then ensures the method label is defined in the class definition. The expression within the method body is converted into its runtime equivalent. As seen by the OpValue active receivers can be removed from a value, apart from *enum* labels, where they are needed to know which object the switch is related to. This is shown by Rule OpChoice which uses the labels to choose the relevant expression branch. Rule OpLabel is used to substitute a recursive expression with the relevant *continue* expression. The OpDelay rule is labelled by a *delay(t)* action and returns a void value *.

A.2.3 Runtime Inference

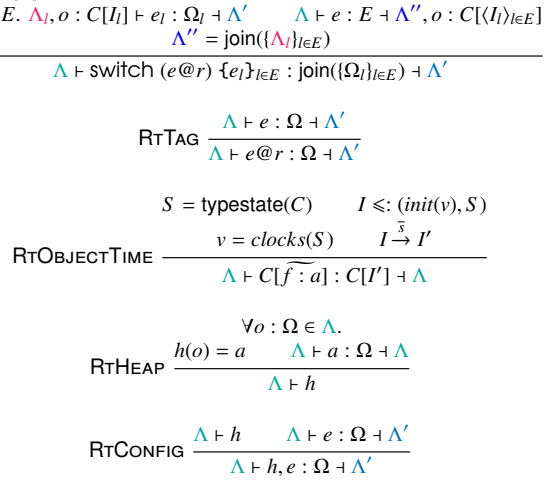

Figure 10: Typestate inference rules for runtime syntax

The extended inference rules for runtime expressions are given in Fig. 10. The only rules shown are the ones that are different from Fig. 6. The RtChoice evaluates to the active receiver instead of a method call. The RtTag infers the typestate of e to infer the typestate of the expression tagged with the active receiver $e@o$. The RtObjectTime is changed such that the declared typestate is augmented with the clock assignment function as similar to the ExpAction rule. All objects on the heap are checked to be consistent by the RtHeap rule. The RtConfig rule types a *configuration* by first inferring the typestate of the expression, which is then used to type the heap.

A.2.4 Typing Context Transitions

We define an extension to the labelled reduction relation found in Mungo for our typing contexts in Fig. 11 which use the same label transitions as our expressions. The rules have been updated with the new inferred typestate. Silent actions leave a typing context unchanged, as shown by the LamId rule. The LamIf states that a context can progress to a subtype of itself. The label involving an assignment of a constant value leaves the typing context unchanged, as shown by the LamTSAsgn rule. The LamCAsgn rule allows objects stored in separate paths to switch their contents, but

$$\textsc{LamId}\ \frac{}{\Lambda \xrightarrow{\tau} \Lambda} \qquad \textsc{LamIf}\ \frac{\Lambda' \leqslant: \Lambda}{\Lambda \xrightarrow{\mathsf{if}} \Lambda'} \qquad \textsc{LamTSAsgn}\ \frac{}{\Lambda \xrightarrow{r.f=c} \Lambda}$$

$$\textsc{LamAction}\ \frac{}{\Lambda, r : C[\{T\ m(T') : I\}] \xrightarrow{r.T\ m\ T'} \Lambda, r : C[I]}$$

$$\textsc{LamCAsgn}\ \frac{}{\Lambda, r.f : C[\mathsf{end}], r' : C[I] \xrightarrow{r.f=r'} \Lambda, r.f : C[I], r' : C[\mathsf{end}]}$$

$$\textsc{LamNew}\ \frac{clocks(S) = v \qquad \mathsf{typestate}(C) = S \qquad (I \leqslant: (init(v), S) \wedge \forall r.f.f' : C'[I'] \in \Lambda.\ I' = \mathsf{end})}{\Lambda, r.f : C[\mathsf{end}] \xrightarrow{r.f.\mathsf{new}\ C} \Lambda, r.f : C[I]}$$

$$\textsc{LamLabel}\ \frac{\Lambda' \leqslant: \Lambda}{\Lambda, r : C[\langle I_l \rangle_{l \in E}] \xrightarrow{r.\langle l' \rangle} \Lambda', r : C[I_{l'}]} \qquad \textsc{LamDelay}\ \frac{\Lambda' = \{r : C[I] \mid r : C[(t, I)] \in \Lambda \wedge t = n\} \cup \{r : U \mid r : U \in \Lambda \wedge U \neq C[I]\}}{\Lambda \xrightarrow{delay(n)} \Lambda'}$$

Figure 11: Reduction relation on typing contexts

requires the original holder of the typestate to be inactive to conform to the linearity properties of Mungo. The LamAction reduces a method-prefixed inferred typestate to typestate continuation when given this method as a label. The LamNew rule allows the inactive typestate to be mapped to the active typestate if it holds under the subtyping relation and all fields with typestates are inactive. A new rule, LamDelay has also been added that states when the typing context contains typestates augmented with a time constant, we can follow the reduction relation which removes these time augments.

A.3 Proof

The proof is by induction on the structure and contexts of the expression e. The reduction is deterministic due to the uniqueness of ℓ, h' and e'.

Two base cases are present when e is of the form $\mathcal{E}[v]$ with $\mathcal{E}$ elementary, but not of the form $\mathcal{E}[\mathcal{E}']$ with $\mathcal{E}' \neq []$, and $\mathcal{E}[e_1]$.

- When e is a value, this case is trivially true
- If e is not a value then it can be any expression contained in the syntax. The most interesting cases are given after the inductive case.

For the inductive case, we have $e = \mathcal{E}[e_1]$ where $\mathcal{E} \neq []$ and e_1 isn't a value.

By assumption, we have $\Lambda \vdash h, \mathcal{E}[e_1] : \Omega \dashv \Lambda'$ and using the inversion of rule RtConfig we have $\Lambda \vdash \mathcal{E}[e_1] : \Omega \dashv \Lambda''$. Using Lemma 8 we can obtain $\Lambda \vdash e_1 : \Omega_1 \dashv \Delta_1$. By the inductive hypothesis, we know there exists h', ℓ such that $h, e_1 \xrightarrow{\ell} h', e_2$. We also know that $\Lambda \xrightarrow{\ell} \Lambda'$ and $\Lambda' \vdash h, e_2 : \Omega_2 \dashv \Delta_1$. We can again use the inversion of rule RtConfig to get $\Lambda' \vdash h$ and $\Lambda' \vdash e_2 : \Omega_2 \dashv \Delta_1$, where $\Omega_2 \leqslant: \Omega_1$. We can then obtain $h, \mathcal{E}[e_1] \xrightarrow{\ell} h', \mathcal{E}[e_2]$ by using rule OpCtx. Lemma 5 allows us to obtain $\Lambda' \vdash \mathcal{E}[e_2] : \Omega' \dashv \Lambda''$ with $\Omega' \leqslant: \Omega$. As we have $\Lambda' \vdash h$ and $\Lambda' \vdash \mathcal{E}[e_2] : \Omega' \dashv \Lambda''$ we use rule RtConfig to obtain our conclusion of $\Lambda' \vdash h', \mathcal{E}[e_2] : \Omega' \dashv \Lambda''$.

We now list the most interesting examples of the base case where e is not a value. For each case, we provide the proof for statement one and statement two for **Theorem 1**.

A.3.1 Case e is ($r.f = \mathsf{new}\ C$)

Statement 1 To obtain $\Lambda \vdash h, e : \Omega \dashv \Lambda''$ we use the inversion of the ExpNew typing rule.

$$\textsc{ExpNew}\ \frac{clocks(S) = v \qquad o \neq \mathsf{this} \qquad I \leqslant: (init(v), S) \qquad \mathsf{typestate}(C) = S \qquad \forall o.f : C'[I'] \in \Lambda \implies I' = \mathsf{end}}{\Lambda, o : C[\mathsf{end}] \vdash o = \mathsf{new}\ C : \mathsf{void} \dashv \Lambda, o : C[I]}$$

This gives us $\Lambda = \Delta_1, r.f : C[\mathsf{end}]$, $\Omega = \mathsf{void}$ and $\Lambda'' = \Delta_1, r.f : C[I]$.

From the hypothesis we know $h, r.f = newC$. We can then use the OpNew to obtain ℓ.

$$h, r.f = \mathsf{new}\ C \xrightarrow{r.f.\mathsf{new}\ C} h\{r.f \mapsto C[f : \widetilde{\mathsf{init}}(T)]\}, *$$

where $h' = h\{r.f \mapsto C[f : \widetilde{\mathsf{init}}(T)]\}$ and $e' = *$.

By ℓ and the LamNew rule we obtain Λ'.

$$\Delta_1, r.f : C[\mathsf{end}] \xrightarrow{r.f.\mathsf{new}\ C} \Delta_1, r.f : C[I] = \Lambda''$$

such that $I \leqslant: (init(v), S)$ and for all fields $r.f.f' : C'[I'] \in \Delta_1$ and state $I' = \mathsf{end}$ where clocks(S) = v and typestate(C) = S. We can perform this reduction of the typing states as the premises are true by the inversion of ExpNew. This gives us $\Lambda' = \Delta_1, r.f : C[I] = \Lambda''$.

We can apply typing rule ExpVoid on e' which is *.

$$\Lambda'' \vdash * : \mathsf{void} \dashv \Lambda''$$

we show $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$ We need to now prove $\Lambda' \vdash h'$ specifically,

$$\Delta_1, r.f : C[I] \vdash h\{r.f \mapsto C[f : \widetilde{\mathsf{init}}(T)]\}$$

By hypothesis $\Delta_1, r.f : C[\mathsf{end}] \vdash h$. Now we want to type the updated reference $r.f$ to $C[f : \widetilde{\mathsf{init}}(T)]$. By rule RtObject and an empty set of labels, $\overline{s}$, we have

$$\frac{S = \mathsf{typestate}(C) \qquad I \leqslant: (init(v), S) \qquad v = clocks(S)}{\Lambda \vdash C[\widetilde{f : o}] : C[I'] \dashv \Lambda}$$

Lemma 4 can then be used to show that the heap update holds under Λ'. RtConfig can be used to show

$$\Delta_1, r.f : C[I] \vdash h\{r.f \mapsto C[f : \widetilde{\mathsf{init}}(T)]\}, * : \mathsf{void} \dashv \Lambda''$$

as required where $\Omega \leqslant: \Omega'$ as $\Omega = \Omega' = \mathsf{void}$.

Statement 2 We have $r : C[end] \in \Lambda$ and $r : C[I] \in \Lambda'$. As $I = \mathsf{end}$ which is an inactive typestate, we can not find a s for a typestate transition, so this property trivially holds.

A.3.2 Case e is $r.f = r'$

Statement 1 To obtain $\Lambda \vdash h, e : \Omega \dashv \Lambda''$ we use the hypothesis and rule RtConfig to get $\Lambda \vdash h$ and $\Lambda \vdash r.f = r' : \Omega \dashv \Lambda''$. By inversion and typing rule ExpTSAsgn

$$\frac{\Lambda \vdash r' : C[I] \dashv \Delta_1, r.f : C[\mathsf{end}]}{\Lambda \vdash r.f = r' : \mathsf{void} \dashv \Delta_1, r.f : C[I]}$$

where $\Omega = \mathsf{void}$ and $\Lambda'' = \Delta_1, r.f : C[I]$, and for readability we let $\Delta_2 = \Delta_1, r.f : C[\mathsf{end}]$.

Let $r' = r.f$. We use ExpTSAsgn with $r.f$ replacing r' as $S = \mathsf{end}$, the derivation holds, and the proof continues trivially.

Let $r' \neq r.f$. Since $r' : C[I]$ in the premise of the above derivation, we know it must be obtained by ExpClass. This rule implies that Λ and Δ_2 only differ by the typing of r'. Therefore we have $\Lambda = \Delta_3, r' : C[I], r.f : C[\mathsf{end}]$.

We can obtain ℓ by the hypothesis and OpTSAssgn rule to get

$$h, r.f = r' \xrightarrow{r.f=r'} h'\{r.f \mapsto h(r')\}, *$$

where $h' = h\{r' \mapsto \mathsf{null}\}$, $\ell = r.f = r'$, and $e' = *$.

By rule LamCAsgn

$$\Delta_3, r' : C[I], r.f : C[\mathsf{end}] \xrightarrow{r.f=r'} \Delta_3, r.f : C[I], r' : C[\mathsf{end}]$$

where $\Lambda = \Delta_3, r' : C[I], r.f : C[\mathsf{end}]$ and $\Lambda' = \Delta_3, r.f : C[I], r' : C[\mathsf{end}]$. Since $\Lambda'' = \Lambda'$, by applying rule ExpVoid we conclude $\Lambda' \vdash * : \mathsf{void} \dashv \Lambda''$, therefore $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$.

We need to no show $\Lambda' \vdash h'$, specifically

$$\Delta_3, r.f : C[I], r' : C[\mathsf{end}] \vdash h'\{r.f \mapsto h(r')\}$$

where $h' = h\{r' \mapsto \mathsf{null}\}$. By the hypothesis we have,

$$\Delta_3, r' : C[I], r.f : C[\mathsf{end}] \vdash h$$

We can use Lemma 4 for $r.f$ and r' to get $\Lambda' \vdash h'$. Finally, we can conclude by RtConfig to obtain $\Delta_3, r.f : C[I], r' : C[\mathsf{end}] \vdash h'\{r.f \mapsto h(r')\}, * : \mathit{void} \dashv \Lambda''$ where $\Omega \leqslant: \Omega'$ as $\Omega = \Omega' = \mathsf{void}$.

Statement 2 We have $\{r' : C[I], r.f : C[\mathit{end}]\} \in \Lambda$ and $\{r.f : C[I], r' : C[\mathit{end}]\} \in \Lambda'$. As similar to the previous case, we can not find a s such that the inactive typestate *end* transitions, so for $r.f : C[\mathit{end}]$ the property holds trivially.

For $r' : C[I] \in \Lambda$ and $r' : C[\mathit{end}] \in \Lambda'$. By the inversion of the RtHeap and RtObject rules, we know that $I_C \leqslant: (\mathit{init}(v_C), S_C)$ and $\exists \overline{s}.\ I_C \xrightarrow{\overline{s}} I$ hold where $S_C = \mathit{typestate}(C)$ and $v_C = \mathit{clocks}(S)$. From the definition of the subtyping relation (Fig. 7), as the entire relation holds for any form of I, we know $I \leqslant: (v, S)$ for some (v, S) and that takes any possible time constraint $\delta \in S$ in a relation's premise must also hold, satisfying time property one.

As $I \leqslant: (v, S)$ as know that if there exists an s that transitions (v, S) to (v', end) we know that from the premise of any possible transition rule given in Definition **??** that clocks are either reset, or stay the same, satisfying time property two.

For (v', end) we know by rule S-End that $\mathsf{end} \leqslant: (v, \mathsf{end})$ for any v, therefore $I' \leqslant: (v', S')$ satisfying time property three.

A.3.3 Case e is r.m(v)

Statement 1 By hypothesis and the RtConfig rule we know $\Lambda \vdash h$ and $\Lambda \vdash r.m(v) : \Omega \dashv \Lambda''$. Can then use the inversion of the ExpAction rule to get

$$\frac{\begin{array}{c} T\ m(T'\ x)\ \{e'\} \in \mathsf{methods}(C) \quad I \coloneqq \mathit{removeDelays}(I', I) \\ \Delta_2, r : C[I'], x : \Omega' \vdash e'\{r/\mathsf{this}\} : \Omega \dashv \Delta_1, r : C[I] \\ \Lambda \vdash e : \Omega' \dashv \Delta_2, r : C[\{T\ m(T') : I'\}] \end{array}}{\Lambda \vdash r.m(e) : \Omega \dashv \Delta_1, r : C[I]}$$

where $\Lambda'' = \Delta_1, r : C[I]$ and let $\Lambda''' = \Delta_2, r : C[\{T\ m(T') : I'\}]$. Then, $\Lambda(r) = \Lambda'''(r) = C[\{T\ m(T') : I'\}]$.

Let $\Lambda = \Delta_3, r : C[\{T\ m(T') : I'\}]$. By hypothesis and by rule OpAction

$$\widetilde{h, r.m(v)} \xrightarrow{r.T\ m\ T'} h, e\{v/x\}\{r/\mathsf{this}\}@r$$

as $h(r) = C[\widetilde{f : o}]$ and $T\ m(T'\ x)\ \{e\} \in \mathsf{methods}(C)$ from the inverse of RtObject and ExpAction. We obtain $h' = h$, $\ell = r.T\ m\ T'$, and $e' = e\{v/x\}\{r/\mathsf{this}\}@r$

We can use LamAction and ℓ to obtain

$$\Delta_3, r : C[\{T\ m(T') : I'\}] \xrightarrow{r.T\ m\ T'} \Delta_3, r : C[I']$$

This gives $\Lambda' = \Delta_3, r : C[I']$

We need to prove $\Delta_3, r : C[I'] \vdash h'$ and

$$\Delta_3, r : C[I'] \vdash e\{v/x\}\{r/\mathsf{this}\}@r : \Omega \dashv \Lambda''$$

We will start with $\Delta_3, r : C[I'] \vdash h'$.

From the hypothesis we know $\Delta_3, r : C[\{T\ m(T') : S\}] \vdash h$. From the inverse of RtHeap we have

$$\frac{h(r) = C[\widetilde{f : o}] \qquad \Lambda \vdash C[\widetilde{f : o}] : C[\{T\ m(T') : I'\}] \dashv \Lambda}{\Delta_3, r : C[\{T\ m(T') : I'\}] \vdash h}$$

and by inversion of rule RtObject on the right-hand side premise we have

$$\frac{\begin{array}{c} I_C \leqslant: (\mathit{init}(v), S) \qquad S = \mathit{typestate}(C) \\ v = \mathit{clocks}(S) \qquad \exists \overline{s}.\ I_C \xrightarrow{\overline{s}} \{T\ m(T') : I'\} \end{array}}{\Lambda \vdash C[\widetilde{f : o}] : C[\{T\ m(T') : I'\}] \dashv \Lambda}$$

We can then perform another reduction with label ℓ, $T\ m(T)$ to obtain

$$\frac{\begin{array}{c} I_C \leqslant: \mathit{init}(v), S \qquad S = \mathit{typestate}(C) \\ v = \mathit{clocks}(S) \qquad \exists \overline{s}.\ I_C \xrightarrow{\overline{s}} \{T\ m(T') : I'\} \xrightarrow{T\ m(T')} I' \end{array}}{\Delta_3, r : C[I]' \vdash C[\widetilde{f : o}] : C[I]' \dashv \Delta_3, r : C[I]'}$$

We use RtHeap to obtain $\Delta_3, r : C[I]' \vdash h$ as required.

We will now show that $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$ holds. By the RtTag rule, we only need to show $\Delta_3, r : C[I'] \vdash e\{v/x\}\{r/\mathsf{this}\} : \Omega \dashv \Lambda''$. By the inverse of ExpAction we have,

$$\Delta_2, r : C[I'], x : \Omega' \vdash e\{r/\mathsf{this}\} : \Omega \dashv \Delta_1, r : C[I]$$

and

$$\Delta_3, r : C[\{T\ m(T') : I'\}] \vdash v : \Omega' \dashv \Delta_2, r : C[\{T\ m(T') : I'\}]$$

We use Lemma 6 to obtain

$$\Delta_3, r : C[I'] \vdash e\{v/x\}\{r/\mathsf{this}\} : \Omega \dashv \Delta_1, r : C[I]$$

as we know either $\Delta_2 = \Delta_3$ with $\Delta_2(v) = \Delta_3(v) = \Omega'$, or $\Delta_2(v) = \Omega''$ and $\Delta_3 = \Delta_2\{v : \Omega'/v : \Omega''\}$. As $\Omega' = \Omega$, we have $\Omega' \leqslant: \Omega$ as required.

As we have shown $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$ and $\Lambda' \vdash h'$ we use the RtConfig rule to conclude and obtain

$$\Delta_3, r : C[I'] \vdash h, e\{v/x\}\{r/\mathsf{this}\} : \Omega \dashv \Delta_1, r : C[I]$$

as required.

Statement 2 We have $r : C[Tm(T') : I'] \in \Lambda$ and $r : C[I'] \in \Lambda'$. By the inverse of the RtObject rule above, we know that $I_C \leqslant: (init(v), S)$ holds where $S = typestate(C)$ and $v = clocks(S)$. From the definition of the subtyping relation (Fig. 7) we know for this method call that either S-Method or S-Method-Time relation holds. Therefore, we have a (v, S) in the form of $(v,\ T\ m(T') : S)$ or $(v, [\delta, P]\ T\ m(T') : S)$. For the first form of (v, S). We have the transition of $Tm(T')$ as a transition to the form (v', S), as this transition does not have any time constraints in the premise, the first property trivially holds.

The second property also holds as $v = v'$ so no clock changes occur from the premise of the transition, so we have $\forall \pi. v'(\pi) \geq v(\pi)$.

For the second form of (v, S) we have $(v, [\delta, P]\ T\ m(T') : S)$ we transition using the label s of $Tm(T')$ to obtain (v', S). As we know that the subtyping relation holds for $I_C \leqslant: (init(v), S)$. We know this transition from (v, S) to (v', S') also holds. In this case it means the premise of our transition also holds which means the time constraints δ present must hold under v, so we have $v \models \delta$ as required for time property one.

From the premise of this transition, we know that clocks within the reset predicate are reset to 0. If $P \neq \emptyset$ then at least one $\pi \in v$ is reset to zero, so the second property holds as $\exists x. v'(x) = 0$. If $P = \emptyset$ then $\forall x. v'(x) \geq v(x)$ then the second time property holds also.

For both of the forms we have that (v', S') holds under the subtyping relation as we know $I_C \leqslant: (init(v), S)$. As $Tm(T') : I'$ transitions to I' for both forms, and we know that $I' \leqslant: (v', S')$ in both cases we have that time property three holds also.

A.3.4 Case *e is* switch $(l'@r)\ \{e_l\}_{l \in E}$

Statement 1 We use the hypothesis and RtConfig to obtain Λ and Λ'' as follows: $\Lambda \vdash h$ and $\Lambda \vdash \text{switch}\ (l'@r)\ \{e_l\}_{l \in E} : \Omega \dashv \Lambda''$.

By inversion and rule RtChoice

$$\frac{\forall l \in E \quad \Delta_l, r : C[I_l] \vdash e_l : \Omega_l \dashv \Lambda'' \qquad \Lambda \vdash l' : E \dashv \Delta_1, r : C[\langle l : I_l \rangle_{l \in E}] \quad \Delta_1 = \biguplus_{l \in E} \Delta_l}{\Lambda \vdash \text{switch}\ (l'@r)\ \{e_l\}_{l \in E} : \text{join}(\{\Omega_l\}_{l \in E}) \dashv \Lambda''}$$

where $\Omega = \text{join}(\{\Omega_l\}_{l \in E})$. We obtain $\Lambda = \Delta_1, r : C[\langle l : I_l \rangle_{l \in E}]$ and $\Lambda'' = \Lambda''$.

By hypothesis and by rule OpChoice

$$h, \text{switch}\ (l'@r)\ \{e_l\}_{l \in E} \xrightarrow{r.\langle l' \rangle} h, e_{l'}$$

for some $l' \in E$, such that $h' = h$, $\ell = r.\langle l' \rangle$, and $e' = e_{l'}$.

By LamLabel and ℓ

$$\Delta_1, r : C[\langle I_l \rangle_{l \in E}] \xrightarrow{r.\langle l' \rangle} \Delta_{l'}, r : C[I_{l'}]$$

we obtain $\Lambda' = \Delta_{l'}, r : C[I_{l'}]$ where $l' \in E$ and $\Lambda' \leqslant: \Lambda$ from the premise of RtChoice. $\Omega = \text{join}(\{\Omega_l\}_{l \in E})$ and $\Omega' = \Omega_{l'}$ therefore by Lemma 7 we have that $\Omega_{l'} \leqslant: \text{join}(\{\Omega_l\}_{l \in E})$.

We now need to show $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$ and $\Lambda' \vdash h'$. By the premise of ExpChoice we know that $\Delta_{l'}, r : C[I_{l'}] \vdash e_{l'} : \Omega_{l'} \dashv \Lambda''$ holds for $l' \in E$. We only need to show that $\Delta_{l'}, r : C[I_{l'}] \vdash h$. By the hypothesis, we have $\Lambda \vdash h$. We can use the inverse of RtHeap and RtObject

$$\frac{I_C \leqslant: (init(v), S) \qquad S = typestate(C) \qquad v = clocks(S) \qquad \exists \overline{s}.\ I_C \xrightarrow{\overline{s}} \langle l : S_l \rangle_{l \in E}}{\Lambda \vdash C[\widetilde{f : o}] : C[\langle l : I_l \rangle_{l \in E}] \dashv \Lambda}$$

to show that there exist $\overline{s}$, such that $I_C \leqslant: (init(v), S)$, S =typestate(C), v=clocks(S), and $I_C \xrightarrow{\overline{s}} \langle l : S_l \rangle_{l \in E}$ and

$$\Lambda \vdash C[\widetilde{f : o}] : C[\langle l : I_l \rangle_{l \in E}] \dashv \Lambda$$

By the typestate transitions, in Definition **??**, we have $\langle l : I_l \rangle_{l \in E} \xrightarrow{l'} I_{l'}$. By applying rule RtObject on this reduction, we have

$$\Delta_{l'}, r : C[I_{l'}] \vdash C[\widetilde{f : o}] : C[I_{l'}] \dashv \Delta_{l'}, r : C[I_{l'}]$$

We conclude by rules RtHeap to obtain $\Lambda' \vdash h'$ and can then conclude using RtConfig to get $\Delta_{l'}, r : C[I_{l'}] \vdash h, e_{l'} : \Omega_{l'} \dashv \Lambda''$ as required.

Statement 2 We have $r : C[\langle I_l \rangle_{l \in E}] \in \Lambda$ and $r : C[I_{l'}] \in \Lambda'$. By the inversion of the RtObject rule, we know that $I_C \leqslant: (init(v), S)$ holds where $S = typestate(C)$ and $v = clocks(S)$. From the definition of the subtyping rules we know that S-Enum holds, meaning (v, S) is of the form $(v, E\ m(T) : \langle \ulcorner [\delta_l, P_l] \urcorner\ S_l \rangle_{l \in E})$. The s for this form is one of the branch labels l' and will transition (v, S) into (v'_l, S'_l). As we know this subtyping relation and S-Enum will have been the rule to obtain this subtyping, we know that every optional time constraint δ_l must have held for v therefore time property one holds.

Following similar logic to the call case above, we know that when we transition with a label l' that some clocks will either be reset or no clocks will be reset, depending on the reset predicate P_l for that label. In this case, time property two holds.

As the subtyping relation holds we know that the premise to S-Enum holds also meaning for every label l', $B_l \leqslant: (v', S')$ holds, therefore for any $I_{l'} \in \Lambda'$ $I_{l'} \leqslant: (v'_l, S'_l)$ also holds for time property three.

A.3.5 Case *e* = *delay*(*t*)

By the hypothesis and the typing rule RtConfig we have $\Lambda \vdash h$ and $\Lambda \vdash delay(t) : \Omega \dashv \Lambda''$. By inversion and typing rule

ExpDelay we have:

$$\frac{\begin{array}{c}\Lambda = \{r : C[(t,I)] \mid r : C[I] \in \Delta_1 \wedge I \neq \mathsf{end}\} \\ \cup\{r : \Omega \mid r : \Omega \in \Delta_1 \wedge \Omega \neq C'[I']\} \\ \cup\ \{r : C''[end] \mid r : C''[end] \in \Delta_1\}\end{array}}{\Lambda \vdash delay(t) : \mathsf{void} \dashv \Delta_1}$$

where $\Lambda = \Lambda$ and it contains r of the form $r : C[(t,I)]$ and $\Lambda'' = \Delta_1$ where it contains $r : C[I]$ for every $r : C[(t,I)]$ in Λ.

By the hypothesis and typing rule OpDelay we have:

$$h, delay(t) \xrightarrow{delay(t)} h, *$$

such that $h' = h$,$e' = *$, and $\ell = delay(t)$.

Using ℓ and Ty-Delay we have $\Lambda \xrightarrow{delay(t)} \Lambda'$ where Λ' contains $r : C[I]$ for every $r : C[(t,I)]$ in Λ. Meaning $\Lambda'' = \Lambda'$.

By applying rule ExpVoid on e' we have $\Lambda'' \vdash * : void \dashv \Lambda''$. This means $\Lambda' \vdash e' : \Omega' \dashv \Lambda''$ holds, where $\Omega' = \Omega = \mathsf{void}$. It remains to prove $\Lambda'' \vdash h'$ specifically, $\Lambda'' \vdash h$. By the hypothesis, we know $\Lambda \vdash h$. By rule RtHeap we have

$$\frac{h(r) = C[\widetilde{f : o}] \qquad \begin{array}{c}\forall r : C[(t,I)] \in \Lambda. \\ \Lambda \vdash C[\widetilde{f : o}] : C[(t,I)] \dashv \Lambda\end{array}}{\Lambda \vdash h}$$

We can then take an arbitrary r and apply the inverse of the RtObject rule

$$\frac{\begin{array}{cc} I_C \leqslant: (init(v), S) & S = typestate(C) \\ v = clocks(S) & \exists \overline{s}.\ I_C \xrightarrow{\overline{s}} (t,I)\end{array}}{\Lambda \vdash C[\widetilde{f : o}] : C[(t,I)] \dashv \Lambda}$$

We can perform another reduction with ℓ, $(delay(t))$ to obtain:

$$\frac{\begin{array}{cc} I_C \leqslant: (init(v), S) & S = typestate(C) \\ v = clocks(S) & \exists \overline{s}.\ I_C \xrightarrow{\overline{s}} (t,I) \xrightarrow{delay(t)} I\end{array}}{\Lambda''' \vdash C[\widetilde{f : o}] : C[I] \dashv \Lambda'''}$$

We know that Λ''' is Λ'' due to the transition of $delay(t)$ which changes all $r : C[(t,I)]$ to $r : C[I]$ which is the definition of Λ'' in the Delay rule. We now use the Heap rule with $\Lambda''' \vdash C[\widetilde{f : o}] : C[I] \dashv \Lambda'''$ as the premise. As r was arbitrary, we know that the previous step can be applied to all $r \in \Lambda$ to get $\Lambda'' \vdash C[\widetilde{f : o}] : C[I] \dashv \Lambda''$ to match the premise of the RtHeap rule.
We can then conclude with RtHeap and RtConfig to obtain $\Lambda'' \vdash h, * : void \dashv \Lambda''$.

Statement 2 For every $r : C[(t,I)] \in \Lambda$ we have $r : C[I] \in \Lambda'$. Every $r \in \Lambda$ is true under the RtObject rule meaning its premise holds. Therefore, we can find $(t,I) \leqslant: (v,S)$ for every $r \in \Lambda$. We can progress using $s = t$ to get $I \leqslant: (v + t, S)$. We know that the progression had no time constraints δ from the premise of the StTime rule therefore time property one holds.

As $v' = v + t$ for every progression, we know that the second time property holds as $\forall x.v'(x) \geq v(x)$ as $t \in \mathbb{Q}_{\geq 0}$.

From the premise of the subtyping relation of S-Time we know that $I \leqslant: (v + t, S)$ as this is the form of the second $I' \in \Lambda'$ we know that for every r time property three holds.

As we have proved the statements for all base cases, we know the overall theorem holds. □

B. IMPLEMENTATION USING JASTADD

JastAdd is an open source meta compiler that makes use of reference attribute grammars written using a declarative specification. The main feature of JastAdd is the ability to declaratively define *attributes* for various nodes within an abstract syntax tree. These attributes can range from simple fields, such as Integer or Strings, to references to other nodes within the tree, allowing graph properties to be defined. In JastAdd the nodes of the syntax tree are represented as objects, with attributes represented as methods of these nodes. This changes a simple AST into a graph model with object-oriented features. Additionally, the programmer can create standard Java classes that can be used within node attributes to calculate or contain specific values held within *aspects*. There are two main types of attributes *synthesised* and *inherited*.

Synthesised attributes propagate information downward in the AST. Following the object-oriented style of JastAdd this is equivalent to each of the node's subclasses being assigned this attribute that they can overwrite. For example,

```
syn boolean ANode.Attr(String str);
```

defines an Attr attribute that must be implemented by all nodes that are instances of ANode or children nodes located at this subtree. Also note that this attribute has parameters. This provides an object-oriented style to these attributes, which allow more complex equation values.

Inherited attributes propagate information upwards in the AST. These attributes require one of the ancestor nodes to provide an equation for this an attribute. For example,

```
inh ANode BNode.getAncestor();
eq ANode.getBNode(int i).getAncestor() =
    this;
```

states that an ancestor node of BNode must provide an equation for the getAncestor() attribute. This is captured by the next equation which is declared on the ANode node, where it provides itself as the equation to all child nodes rooted at the BNode subtree. Inherited attributes allow child nodes to ensure they are given information only available in ancestor nodes.

Another important attribute is a *collection* attribute. These attributes have composite values and are similar to synthesised attributes, where child nodes add *contributes* to this collection. An example collection below

```
coll Set<CNode> ANode.getCNodes() [new
    HashSet<CNodes>()] with add root ANode
    ;
CNode contributes this to ANode.getCNodes
    () for getANodeAncestor();
```

This collection rooted at each ANode creates a new HashSet which stores each CNode child node that chooses to contribute to this collection. This contribution can be conditional, and the sub nodes can be located anywhere in the subtree rooted at ANode. A specific contributing method can be declared for the collection object, in this example add has been used.

Mungo makes use of the ExtendJ [2] extensible Java compiler created using JastAdd. It provides an updated AST grammar that defines Typestates following the grammar presented in Section 3.1. Relevant attributes are defined for the existing Java AST nodes that act as the inference system. This inference system walks through all execution paths of a class, before collecting possible semantic errors after comparing the inferred typestates to the collected declared typestate. A new GraphNode class is created which acts as a Java class which stores information about methods that have been called on a specific execution path. When analysing the Java classes, a GraphNode is created for each execution path which is compared to the parsed Typestate declaration and compared using a Graph algorithm following the subtyping relation.

The time extension of Mungo was implemented by extending the Mungo implementation written in JasAdd using the ExtendJ compiler. Updates were provided to the Typestate grammar. This grammar involved the addition of three new node types, *TimeConstraint*, *ResetPredicate*, and *ClockConstraint*. Each node type corresponds to the grammar of declared typestates shown in Section 3.1. Each separate type of clock constraint is given its own node type to allow a polymorphic system to be used for constraint checking, as described below. ExtendJ uses the Beaver parser [1] to parse source files and create the relevant JastAdd nodes. The example below shows how Beaver is used to distinguish between individual instances of clock constraints while returning the relevant JastAdd node class.

```
ClockConstraint c_constraint =
    c_constraint.left OR c_constraint.
    right
{: return new OrCond(left, right); :}
```

A Time aspect was created to define specific attribute definitions relevant to time constraints. A TypestateDecl.getClocks() collection attribute was defined to relate to the *clocks* function described found in Fig. 13 which allows the typestate definition to contain the set of all clock names defined in a typestate. Below is a listing of the Delay expression rule.

```
eq Delay.getGraph(Set<TypestateVar> env){
    GraphNode n = new DelayNode(getExpr())
        ;
    for(TypestateVar v: env){
        v.current = n.addNext(v.current);
            }
    return true; }
```

The getGraph attribute is central to Mungo's implementation of the inference algorithm to detect the execution paths of an object in the source code. This equation creates a new DelayNode which is then added to all current Typestate paths held within the environment. This ensures that a delay expression updates known inferred typestate classes.

A synthesised attribute of *ConstraintHolds(HashMap<String, Double> clocks)* is defined for all instances of ClockConstraint node. This follows a polymorphic style of method definition, where all instances provide an equation stating the conditions for the constraint to hold. For *AndCond*, *OrCond*, *NotCond*, and *ParenCond* it recursively accesses the equation for each sub clock constraint held within this node.

GraphNode has been updated to include time constraints for relevant method nodes, following the original graph algorithm, which checks if an inferred typestate is included in the declared typestate. Following the subtyping relation presented in Section 3.5 the algorithm now contains a set of clocks and their current time value. When comparing a MethodNode found in an inferred typestate to its declared equivalent, the time constraints are checked to ensure they hold for this method execution following the synthesised attribute described above, otherwise a semantic error is collected.

A new DelayNode is defined which is used to store delay expressions in typestate paths. During a subtype check, these delay nodes are processed and update the clock values held by the declared typestate. This allows the delay nodes to propagate time forward through the subtyping check while being reset by relevant reset predicates when present. This allows the delay nodes to be individual nodes in the graph that don't need information about the current timing of methods call that may have occurred due to previous nodes. Delays nodes are also only propagated forward for specific branches, this ensures relevant recursive and branch features hold for the implementation.

Delays held within external methods is an important feature of our typing system. The original Mungo system forbids aliasing of typestate defined classes. This is captured in the call inference rule, which checks the receive typestate is unchanged by the method call. Our extension to the typing system has the same constraint but allows delay calls to be added to this inferred typestate. This allows methods to only update the recorded time in a method call without changing typestate method paths. This is captured in our implementation by defining a new attribute for MethodDecl nodes that collect all possible tsDelay expressions found within the definition. During the inference of a method call, these delays are accessed and inferred for this method path.

C. SUBTYPING EXAMPLE

An example of the subtyping relation tree is given below. We first present the typestate definition, which will act as our declared subtype, *S*.

```
1 typestate MessageStackProtocol {
2  Stack = {Check checkForMessage(): <
       RECEIVED: [xm<10, {xm}] Present,EMPTY
       : [xm<10] EndState>}
3  Received = {[xm < 10, {xm}] void
       answerMessage(): Stack,[xm < 5] void
       ignoreMessage(): Stack}
4  EndState = {void shutDown(): end}
5  Present = {[xm < 2] Message getMessage():
        Received} }
```

Given the inferred subtype *I* below, we obtain the follow-

ing subtyping relation.

I = ms:MessageStack[μLoop.{Check checkForMessage() :
{String getMessage:(3, {void answerMessage:Loop}),
(4, {void shutdown:end})}]

$$\dfrac{\dfrac{\dfrac{\dfrac{(1.1)\qquad (1.2)}{\text{Check cFM:...} \leqslant: (\{xm:0\}, \text{Check cFM...})}}{(1)\qquad \{\text{Check cFM:...}\} \leqslant: (\{xm:0\}, \{\text{Stack}\})}}{\Upsilon \cup \{\mu Loop.I', (\{xm:0\}, \text{Stack})\} \vdash I'\{\mu Loop.I'/Loop\} \leqslant: (\{xm:0\}, \text{Stack}) \qquad (1)}}{\mu Loop.I' \leqslant: (\text{init}(\text{clocks}(\text{Stack})), \text{Stack})}$$

S-Rec1
S-Branches
S-Enum

This relation begins with the *Loop* label held within I being replaced with a copy of $\mu X.I'$. The StBranches rule is then used to select the correct method, which is then matched StEnum. The two separate relations for both enum branches are given in (1.1) and (1.2) respectively.
(1.1):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\mu Loop.I', (\{xm:0\}, \text{Stack})\} \in \Upsilon}{\mu Loop.I' \leqslant: (\{xm:0\}, \text{Stack}) \qquad xm:3 \models \text{xm<10}}}{xm \to 0}}{\text{void answerMessage:}\mu Loop.I' \leqslant: (\{xm:3\}, \textit{Received})}}{(3, \{\text{void answerMessage:}\mu Loop.I'\}) \leqslant: (\{\text{xm:0}\}, \text{Received})}}{xm:0 \models \text{xm<2}}}{\{\text{String getMessage:(3,...)}\} \leqslant: (\{\text{xm:0}\}, \text{Present})}$$
$$xm:0 \models \text{xm<10}$$

S-Method-Time
S-Time
S-Method-Time
S-Terminate

This relation captures the subtyping of the RECEIVED label branch. The starting StMethod holds as the clock constraints hold for the initial xm value of 0. This clock is then updated using the StTime, which again holds for the next StMethod rule above. This clock value is then reset, which allows the StTerm rule to hold as this recursive relation was added to the set Υ at the beginning of our derivation. (1.2):

$$\dfrac{\dfrac{\dfrac{}{end \leqslant: end}\ \text{S-End}}{\text{void shutdown:end} \leqslant: (\{xm:4\}, \text{EndState})}\ \text{S-Method}}{(4, \{\text{void shutdown:end}\}) \leqslant: (\{xm:0\}, \text{EndState})}\ \text{S-Time}$$

The above derivation is for the EMPTY label branch and involves a simple StMethod followed by the update of clocks using the StTime rule, finished by a StEnd rule.

D. EXTRA FIGURES

$$\begin{aligned}
convert(\ulcorner \kappa \urcorner\ T\ m(T) : S) &= \{T\ m(T) : convert(S)\} \\
convert(E\ m(T) : \langle \ulcorner \kappa_l \urcorner\ S_l \rangle_{l \in E}) &= \{E\ m(T) : \langle S_l \rangle_{l \in E}\} \\
convert(\widetilde{G}) &= \bigcup_{G \in \widetilde{G}} convert(G) \\
convert(X) &= X \\
convert(\mu X.S) &= \mu X.convert(S)
\end{aligned}$$

Figure 12: Convert function

$$\begin{aligned}
clocks(\widetilde{G}, C) &= \bigcup_{G \in \widetilde{G}} clocks(G, C) \\
clocks([\delta, P]\ T\ m(T') : S, C) &= \begin{cases} fn(\delta) \cup clocks(S, C \cup \{S\}) & \text{if } S \notin C \\ fn(\delta) & \text{otherwise} \end{cases} \\
clocks(E\ m(T) : \langle [\delta_l, P_l]\ S_l \rangle, C) &= fn(\delta_l) \\
&\quad \cup \bigcup_{l \in E} \{clocks(S_l, C \cup \{S\}) \mid l \in E \wedge S_l \notin C\} \\
clocks(S, \emptyset) &= clocks(S, \{S\}) \\
clocks(X, C) &= \emptyset \\
clocks(\mu X.S, C) &= clocks(S, C)
\end{aligned}$$

Figure 13: Clocks function

$$\begin{aligned}
rdy(\widetilde{G}) &= \bigcup_{G \in \widetilde{G}} rdy(G) \\
rdy([\delta, P]\ T\ m(T') : S) &= \{\{\delta\}\} \\
rdy(E\ m(T) : \langle \ulcorner [\delta_l, P_l] \urcorner\ S_l \rangle) &= \{\ \bigcup_{l \in E} \{\delta_l\}\ \} \\
rdy(X) &= \{\} \\
rdy(\mu X.S) &= rdy(S)
\end{aligned}$$

Figure 14: Ready function

$$removeDelays((t, I'), I) = \begin{cases} removeDelays(I', I) & \text{if } noOfDelays((t, I')) \\ & > noOfDelays(I) \\ (t, I') & \text{otherwise} \end{cases}$$
$$removeDelays(I', I) = I'$$

Figure 15: Remove delay function

$$noOfDelays(I) = \begin{cases} 1 + noOfDelays(I') & \text{if } I = (t, I') \\ 0 & \text{otherwise} \end{cases}$$

Figure 16: NoOfDelays function

$$\textsc{EqStart}\ \frac{\emptyset \vdash I \coloneqq I'}{I \coloneqq I'} \qquad \textsc{EqEnd}\ \frac{}{\Upsilon \vdash \mathsf{end} \coloneqq \mathsf{end}} \qquad \textsc{EqTerm}\ \frac{(I, I') \in \Upsilon}{\Upsilon \vdash I \coloneqq I'} \qquad \textsc{EqMethod}\ \frac{\Upsilon \vdash I \coloneqq I'}{\Upsilon \vdash T'\ m(T) : I \coloneqq T'\ m(T) : I'} \qquad \textsc{EqRec1}\ \frac{\Upsilon \cup \{(I, \mu X.I')\} \vdash I \coloneqq I'\{\mu X.I'/X\}}{\Upsilon \vdash I \coloneqq \mu X.I'}$$

$$\textsc{EqBranches}\ \frac{\widetilde{B} \neq \emptyset \qquad \forall B \in \widetilde{B}, \exists B' \in \widetilde{B'}.\ \Upsilon \vdash B \coloneqq B' \qquad \widetilde{B'} \neq \emptyset \qquad \forall B' \in \widetilde{B'}, \exists B \in \widetilde{B}.\ \Upsilon \vdash B \coloneqq B'}{\Upsilon \vdash \widetilde{B} \coloneqq \widetilde{B'}} \qquad \textsc{EqTime}\ \frac{\Upsilon \vdash I \coloneqq I'}{\Upsilon \vdash (t, I) \coloneqq (t, I')}$$

$$\textsc{EqEnum}\ \frac{\forall l \in E.\ \Upsilon \vdash I_l \coloneqq I'_l}{\Upsilon \vdash E\ m(T) : \langle I_l \rangle_{l \in E} \coloneqq E\ m(T) : \langle I'_l \rangle_{l \in E}} \qquad \textsc{EqClass}\ \frac{I \coloneqq I'}{C[I] \coloneqq C[I']} \qquad \textsc{EqBot}\ \frac{}{\mathsf{bot} \coloneqq U} \qquad \textsc{EqGrnd}\ \frac{U \in \{E, \mathsf{bool}, \mathsf{void}\}}{U \coloneqq U} \qquad \textsc{EqEmpty}\ \frac{}{\emptyset \coloneqq \Lambda} \qquad \textsc{EqLambda}\ \frac{\Lambda \coloneqq \Lambda' \qquad U \coloneqq U'}{\Lambda, r : U \coloneqq \Lambda', r : U'}$$

$$\textsc{EqLabel}\ \frac{\Lambda \coloneqq \Lambda'}{\Lambda, \lambda : X \coloneqq \Lambda', \lambda : X}$$

Figure 17: Equivalence relation for inferred typestates (symmetric rule EqRec2 omitted)