Time for Typestates

Luke Gall

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Masters project proposal

6/12/21

Contents

1 Introduction

Typestates [27] are a programming language concept that types the set of methods that an object may perform based on its *state*. After an object calls a method, it's typestate will change to a new state, represented by another typestate. This type system can remove method ordering errors where a programmer may call a method in the wrong state of a class, such as trying to read a File object that hasn't been opened.

Session types [16, 16, 17] are communication protocols that describe a sequence of communications between two or more processes. They provide type safety and deadlock freedom in the concurrent setting and have been adapted to a variety of modern programming languages.

Timed Automata [2] is model that accepts an infinite set of events annotated with time events, clocks, and clock constraints. Events are labelled with the time the event occurred while a set of independent clocks can capture and hold different values of time. Using these clocks, constraints can be added to transitions for this system, adding time-based constraints to actions. Timed Automata has lead to large research field around modelling real-time systems including Communicating Timed Automata [21] and Timed Multiparty Session Types [6]. These systems can be used to model time-sensitive systems and protocols including airplanes, toasters, and certain SMTP protocols [19].

This project aims to research the possibility of extending Typestates with time constraints to model time sensitive Typestates. Drawing upon necessary theory, this project aims to bridge the gap between session types, timed models, and Typestates to extend Mungo [20], a typestate toolchain based on session types, to model time-sensitive methods in Java.

2 Statement of Problem

Research Question Can we extend the Mungo tool and it's typestate theory with time to guarantee typestate conformance and completion under time constraints?

The Mungo tool can statically typecheck and validate protocol confirmation of classes against their typestate definitions. This tool can be used by programmers to typecheck and enforce communication protocols and the correct sequence of methods calls by objects. Many objects have implicit state and a correct ordering of method calls that can be ill defined in code or API documentation. This implicit state can lead to runtime errors that are difficult and expensive to find and debug as they can present themselves in a different area of the code to where the method was called.

Real time systems depend on time constraints between processes in order to correctly match expectations users have on systems. For example in the definition of the SMTP server "An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender". If a programmer was to implement this server in Java, they would have no static tool to ensure they conformed to this time requirement.

The aim of this project is to translate the benefits of typestates and static protocol conformation to the area of real-time systems and protocols. This would allow Mungo to ensure object usages conform to their typestate definition and timing constraints. Allowing a programmer to create a set of classes free from method sequencing, communication, and time based errors.

3 Background Survey

3.1 Session Types and their implementation in programming languages

Introduction Session types are communication protocols that describe a sequence of communications between one or more parties. They codify the commutation protocol between two or more processes and allow for type safety and deadlock freedom in the concurrent setting. Session types were originally developed from the Types for Dyadic Interaction [15] paper presented by Honda. These dyadic types involve two participants where each act as a "co type" with each other in order to model two concurrent types that interact with each via sends, receives, branches, and selections. The paper presents theory that forms the foundation of the Session types research area including substitution within types, reduction relations between types modelling communication, and progress.

Sessions In their next work, Honda [16] defines the first example of *Sessions*. They are a chain or sequence of binary interactions that when combined in a set can model a program. Communication between two parties via a Session create a private port called a *channel* where the messages can be privately passed to the two participants. Their system also includes sends and receives as before but also includes branches of different send options for a session as well as session *delegation*. Delegation models the situation where a process needs to send a partial session type to another process which will continue where the session left off. The paper also presents *duality* which matches the notion of *co types* from Honda [15] where actions in a session type can be matched by its *dual*, this is a process that uses the symmetrical session type. This dual of a session type is where each action in the sequence matches with its symmetrical action, i.e. sends match with receives of the same type, branches match with selections etc. Duality is used in the type system to ensure progress is made and a type system is error free. This simple yet powerful typing system leads to error free typing in the binary and synchronous setting. The typing system is not usable for the asynchronous communication setting, a clear area of future work. The authors also noted that the type system could create complex Session types that couldn't be easily matched by programmers, leading to more complex software design that would be difficult to understand.

Multiparty Asynchronous Session Types The next seminal work in session types was the introduction of *Multiparty Asynchronous Session Types* [17] by Honda et al. This work presented a new versions of session types that could have more than two participants communicating with each other. This extension more closely matches the communication patterns seen in modern communication and concurrent applications. The fundamental properties present in binary session

types were proved for this extension with a powerful type system centered around global and local session types. The notion of channels is extended where fresh channels are created for any two parties that need to communicate to each other. By providing private channels, asynchronous messages between participants don't interfere with other messages from different parties. To eliminate race conflicts linearity of channels is imposed in the global type. This checks that when a common channel is used in two asynchronous actions, the sends and receives must have a temporal order. Session types are written by first creating a *global type* where all communication sequences between the various parties are defined. This global type is then projected onto individual *local types* for each participant where each participant follows the sequence of communications relevant to itself.

Distributed Object Oriented Language with Session Types Session types have been added to different programming languages with object oriented (OO) languages being a large area of work. Ciancaglini et al[12] presented one of the first examples of augmenting an object-oriented language with session types and communication primitives. Their system, Ldoos, denotes the channel usages as the type of a channel within the class definition i.e. *!int.?Int.end myChan*. This syntax was careful to designed to be easily added to existing object practices and styles that programmers are used to while making session types easier to integrate within existing code bases. This syntax style, while matching OO styles of the time, can quickly become verbose and complicated for more complex channels that a class may use. The session types they adapt to this setting have been simplified as well, with no recursion, branching, or selection. The authors state that while recursion doesn't exist, existing iteration structures within the OO language combined with their session type syntax can still match the expressiveness of more complex session types.

MOOSE Ciancaglini et al [11] present MOOSE, a simple object oriented language that is multi-threaded and augmented with session types. With OO programming style in mind, the authors augment linear channels between classes with (binary) session types. They show that a well-typed MOOSE program has progress and can guarantee freedom from communication errors or starvation on live channels. MOOSE differs from Ldoos by having session types being defined for a channel and then attaching that type to any channel that should match it. This syntax style was more modular and clean compared with Ldoos.

MOOSE<: The authors of MOOSE improved upon their earlier work with MOOSE<: [10]. This paper extends the type system with bounded polymorphism and makes the selection more object orientated. MOOSE<: allows session types such as ?(X<:Image).!X which states that an Image object (or subtype) will be received and then an object of the same class will be sent. They also make selection more object-oriented, so that the branch selection is based on the class of the object being sent instead of labels. They prove the progress property for for Moose<:.

Amalgamating Sessions and Methods In *Amalgamating Sessions and Methods* [8], the authors present a unique approach to combining sessions with object oriented languages, expanding over their earlier work with MOOSE<:. Instead of extending object orientated languages with session

types and primitives. They ”amalgamate” sessions and methods such that classes contain a set of session/methods that send asynchronous messages that contain objects on channels. The response of a session response (which is an object) is then matched similar to case matching and then the session/method continues as an expression. They prove progress and subject reduction for their amalgamation and present a language kernel so that it can be used to create this amalgamation in any object orientated language. While providing a interesting and powerful tool, the syntax and operations of the tool don't match those found in either Session types or Object Orientated languages so may struggle to be easily understood by users familiar with either. This approach still provided value over MOOSE<: by not requiring explicit mentions to communication channels (which they argue is unnatural in a OO setting) and providing polymorphism through generics.

Sessions, from types to programming languages Vasconcelos [29] presents a communication example involving two participants that they convert to three separate session type implementations. One involving *pi calculus*. The second to type a linear multi-threaded functional language that adds the notion of thread forking while needing to remove explicit refers to variables that would capture receives and not explicitly mention the continuation term. The third implementation is for object orientated languages, in which the authors use object references instead of channels. Threads in their implementation interact only using object references, which are checked via session types. This paper showcases three interesting examples of session types for different paradigms, showcasing the use and power of session types over different domains.

Modular Session Types Gay et al [13], present Bica, an extension to Java that allows session types to be consumed by multiple methods instead of just the one. They also extended the current theory around session types for object orientated languages by allowing session types to be attached to the class. This session type was used to specify the sequence of method calls available in different states of the class. The session types of communication channels, similar to previous work in MOOSE, are treated as objects to allow a cleaner integration with the session type of the containing class. The binary session types are defined within a class for the channel fields, while also defining the class session type of the possible method calls in each class state. The authors plan to extend the type system to the multiparty version of session types while exploring methods to remove the current linear object condition that applies to classes.

3.2 Time

Timed Automata We now present the notion of time as first defined by Alur and Dill [2] in which they define *Timed Automata*. They present their model following motivation that existing work focused on presenting automata separate from time, purely focused on communication between systems but not the time it would take to do so. They argue that while these systems have lead to powerful insights, they can not model important communications within systems that are real-time dependent such as toasters or airplanes. Their model focuses on creating an ω-language that accepts an infinite sequence of events augmented with time constraints and resets. They model time as monotonically increasing real values. The model accepts an infinite sequence

of *timed words* which is an infinite set of symbols with real-valued time associated with them. The automata contains a set of states as well as a set of labelled edges denoting the transitions between states that occur when it receives a word as an input. They present the notion of a finite set of clocks. Each clock is named and stores a real-time value. This value increases following the time associated with the set of symbols. These clocks are then used in Boolean conditions that are attached to different events such as 'x = 5 $\wedge$ y >2'. These conditions must be true for the automata to transition along an edge annotated with these conditions. The edges can also be augmented with a reset condition which resets a set of clocks during the transition between states. The clocks are all asynchronously independent from each other and can be reset separately, this allows the automata to model different concurrent delays. Timed Automata are shown to to be able to model different aspects of real time systems including liveness, fairness, and nondeterminism. This work acts as the basis for a large field of research extending a variety of different models with the notion of time and time constraints to simulate real-time systems.

Communicating Timed Automata Krcal and Yi [21] present *Communicating Timed Automata* (CTA) to model systems that send and receive messages over channels while following timing constraints. This model is similar to a network of *Timed Automata* extended with unbounded channels where sending and receiving actions are constrained by the values of clocks. Formally the model contains a tuple of timed automata and unidirectional channels which contains the messages sent from each pair of timed automatas. Each automata has an independent set of clocks, disjoint from other automatas, and a set of accepting states. The transitions of the model rely on the timed automata constraints and resets as well as the transition label sent or receive to a particular channel. Reset predicates are modeled as a set of clocks where when a transition fires, all clocks within the set are reset to zero. The authors then present the semantics of the CTA which contains sends, receives, and the passing of time.

Meeting Deadlines Together Bocchi, Lange, and Yoshida [4] present a paper that studies various properties of Communicating Timed Automata. The definitions from Communicating Timed Automata [21] are adapted in order to be deterministic, directed, and not mixed. They authors define a directed CTA when it "contains only sending / receiving actions to/from the same participant". Mixed is defined a containing a mix of different actions, i,e. sends, receives. With these restrictions the authors state that this type of CTA can be seen as a *local session type* [17]. The authors define the notion of a *Timed Communicating System* (TCS) which contains a finite set of CTAs and queues for their communicating channels. They then define a *Synchronous Transition System* (SCS) which models the synchronous execution paths of a TCS. Using the definitions described above, the authors give a condition for safety within CTAs, *multiparty compatibility*. This condition intuitively states that every message that can be sent can be received by another party and that for a receive state, at least one of its expected messages will be received in the future. By proving that a multiparty compatible SCS is safe and giving an equivalence relationship for a TCS then show how to prove a system of CTAs can have the safety property. Progress is also defined for a system of CTAs to prevent errors where time constraints are false or an unexpected message is on its queue. *Zeno-ness* is when a state can only progress by firing messages infinitely in smaller intervals of time. The authors give conditions to ensure this property is false, *Non-zeno*, for a system of CTAs. The conditions the authors have provided can be adapted to different timed

systems, and a possible area of work bridging the notions of Session types and CTAs with their restricted definitions.

Asynchronous Timed Session Types In *Asynchronous Timed Session Types* [5], the authors present a new behavioural typing system that uses (binary) session types to model time reliant processes. Two new time primitives are added to their calculus. *Timeout receive* which attaches a timeout to a receive action, it can be used to model a non-blocking, infinitely blocking, and bounded blocking timeout. *Delay* is used to model the passing or consummation of time, they suggest it can be used to model time intensive actions. Their syntax of Time session types follow the definitions of time found in [2], where a session is associated with time constraints. Every action that the session type can perform can be guarded by time constraints and a reset predicate. To rule out the formation of junk types the authors introduce formation rules. They define a subtyping relation on the types removed from their queues and environment, which they define as *Simple type configurations*. They use a labelled transition system describing pair of type configurations and their transitions. The subtyping relation uses a simulation relation between two simple type configurations such that for every path in configuration is matched by the other. They give the definition for duality, this definition uses similar duality as seen before with session types and have time and reset constraints matching between *duals*. They prove this duality with a *urgent receive* property which forbids time to pass if a message on a queue can be consumed by a receive, i.e. the receive executes instantly. They also give *Subject Reduction* and *Time Safety*. This paper provides some much needed properties for timed asynchronous (binary) session types including progress, duality, and subtyping however the binary nature of the session types reduces the scope of distributed problems it could model.

Timed π-Calculus The action sends the clock name to allow receiving processes to reason about channel delays. For example it allows a process to check the time a message was in transmit, e.g. (c - t <2). The operational semantics of the processes follow closely to standard π-calculus with actions fired only if they meet their clock constraints. A bisimulation relation is presented for their model as well as algebraic laws. The authors presented this model to extend π-calculus such that it can model "cyber-physical" systems which require the time the systems perform tasks. One area not mentioned by the authors is how this model would be typed. It would be interesting to see if timed session types or ideas from CTA could be used to type this calculus.

Time and Exceptional Behaviour In "Time and Exceptional Behaviour in Multiparty Structured Interactions" [22], the authors study an extension of *Conversation Calculus* [30] with time and exception handling. Conversation Calculus is an extension of π-calculus which models distributed multiparty mediums that participants can communicate within. The approach the authors take is different from that of Timed Automata and papers so far mentioned. They allow processes to model two versions when presenting a conversation channel, *default behaviour* and *compensating behaviour*. The conversation channels that a process declares are annotated with a bounded time value and an abort mechanism. When the time value is above zero, the default behaviour occurs, when the time reaches zero the compensating behaviour is used. This bounded time is to model situations when a process should continue for t time units before changing its session, such as an

online banking application requiring user to login again. By providing a compensating behaviour, the model can handle when time bounds reach zero or the abort action is performed, modelling an exception. Unlike other papers, time is fully local to the channel, with no notion of global time. This paper provided an interesting direction in which time was used for bounded actions but contained no notion of clocks or constraints. They provide examples to show when this system would be practical in use but the model is less general compared to other approaches to modelling time.

Timed Multiparty Session types Bocchi, Yang, and Yoshida [6] present a typing theory for timed communications, based on multiparty session types. The authors extend the global and local session types discussed in Multiparty Session Types [17] with clock constraints as found in *Timed Automata*. Global types define the interactions between two or more processes and the clock constraints on the send and receive sides of an action. The reset predicates that are attached to actions are used to model time constraints that must be satisfied in recursive scenarios as well as constraints that require time since last reset. In order to model asynchronous behaviour, they extend the notion of multiparty session types to include states where a message has been sent but not yet received. The authors provide a labelled transition system that ensures no actions invalid time constraints of any other *ready action*, which is any asynchronous action that can be performed at that point. In order to correctly type recursion processes they introduce the *Infinitely Satisfiable* property which gives conditions for constraints within recursive bodies. If a recursive process if infinitely satisifiable then only the single unfolding needs to be type checked. Time error freedom is proved for their type system as well as progress when the timed global type is *feasible* and *wait-free*. Wait freedom ensures that if a process performs a receive action the message will always be on its queue, and it will not need to wait for a send. A global type is feasible if every partial path can terminate over a sequence of actions. The authors then give a relationship between their constrained timed global types and CTAs. This approach differs from the approach taken by *Asynchronous Timed Session Types* by being multiparty and requiring wait-free protocols. While multiparty session types are more practical for representing real life communication protocols, wait-freedom is a difficult property to enforce in real-life systems with some SMTP protocols not being wait-free.

Timed Session Types Bartoletti, Cimoli, and Murgia [3] present a paper where they provide techniques to decide when a binary timed session type admits a *compliant*. If the session type can admit a compliant type, then a procedure they describe can produce the most precise session type to be a compliant. Two session types are compliant when they are composed together and behave correctly. An untimed binary session type always admits a compliant, its dual, but this is not always the case in the timed setting. For example, !int[x<10].?int[x<5] can't admit a compliant. Determining if two untimed session types are compliant is decidable so the authors explore ways to move this theory into the timed setting. To decide if two timed session types are compliant they present a *kind* system which gives the sets of clock valuations for a type, this can detect if a timed session type admits a compliance. They finally show an encoding to turn a timed session type into a Timed Automata.

Timed Runtime Verification Neykova, Bocchi, and Yoshida [24] present a paper which proposes a dynamic verification framework in python for real-time protocols. The paper builds on work around Multiparty Session Types (MPST) [17] and the Scribble protocol language [31] by extending Scribble with time constraints, clocks, and resets based around Timed Automata. They then present an API for Python that can program timed distributed processes following the Scribble protocols. Scribble protocol language is used to write global session types which can then be checked for well-formedness and then projected onto each individual local session type. The authors extend Scribble by extending participants with clocks and action labels based on clock constraints. Formal semantics are presented to correctly characterise the correct behaviour of a global protocol augmented with time. This ensures ready actions as discussed in Timed Multiparty Session Types aren't invalidated. Two properties from MPST have been extended to this context, *wait-freedom* and *feasibility*. A checker has been included to ensure these properties hold so they can state *progress* for their system. After presenting their timed API for Python the authors touch on a study of various timed patterns they discovered from literature and show how they can be implemented int their extended Scribble language. This paper shows a more practical use of timed session types compared to others in the explored literature and while some constraints are more restrictive than other papers, for example non-waitfree, the examples shown at the end show the real world use of their system.

3.3 Typestates

Introduction Strom and Yemini [27] were the first to introduce the notion and underlying theory of *typestates*. Typestates are a programming language concept to improve reliability by typing the set of operations an object can perform depending on its state. Most objects have implicit state, such as file classes being open or closed. The operations avaliable to the object in each state can be distinct, e.g. can open a file in the Closed state but not read. These implicit states are usually poorly documented or left as comments and the programmer is expected to know which methods can be performed by the object. This can lead to a large amount of "nonsensical" errors which can be large and difficult to find as the error may propagate to other parts of the code. The goal of typestates as introduced by this paper is to remove the errors associated with incorrect ordering of operations on classes by introducing typestates for classes. Each typestate provides a set of operations / functions that the object can perform in this state, and the transitions to different typestates each method has. The compiler can then check that an object only performs a valid operations depending on its typestate. The ideas and theory presented in this paper is used as the base for a wide ranging research field.

Plaid Aldrich et al [1] presented a typestate based programming paradigm in which objects are intertwined with states and the transitions between them. This paradigm was introduced with a new language called *Plaid* which focused on adding typestates at the language level through its syntax rather than a separate checker. Programmers would write the individual states that an object contains, each with their own methods and fields. Methods in these states can then define the transition to a new state in its declaration. The language also uses a permission based aliasing system to ensure the states of parameters aren't lost over aliased objects. The authors presented an

interesting and clean approach to adding typestates to a Java like language, however this paper touched only on the paradigm and didn't provide the fundamental theory for the system or a type checker. Sunshine et al [28] presents an update of their previous work with Plaid by presenting a full language and the core semantics. Incorporating the ideas mentioned in their previous work, the authors believe that a typestate oriented programming model can lead to a variety of benefits including more consise programmers with less defensive code, reflective code based on a stateful design and more. The main advantage this language has over other other approaches is having the states at language level, removing the need for an external checker. The language is powerful and modern, targeting the JVM and providing first class generators, functions, and complex state combinations to model various *Statecharts*, a visual formalism [26]. The language supports substates, similar to subclasses in Java, which allows different "dimensions" of states to be held by objects allowing complex states to be modelled. The main negative of the language is the lack of static checking. Dynamic checking has been provided but static checks offered by other projects may be more desirable to programmers.

Chemical approach to typestates A novel approach to typestates was presented by Crafa and Padovani [9] where they modelled their system through a Chemistry metaphor. In this metaphor, the states and operations of objects are like molecules of messages. The transformation of states are like chemical reactions. Their system is inherently concurrent and allows several processes to change and access different objects. Following the ideas of typestates, an object can present different methods available depending on its state. Their system is based on *Join Calculus* and involves a set of objects and a "chemical soup" which is a set of messages depicting the state and operations that want to be performed on the set of objects. The messages in the soup combine into a "complex molecule" which contains the state of an object and an operation that should be performed on it. The objects are written with guards to match these molecules and then progress into a different state by releasing a new molecule / message. Their presented type system can enforce constraints on the objects, including the message types accepted by the object depending on its state. The private messages of these objects are state representations whereas public messages are operations that are to be performed on them. The authors wish to perform further work onto their elegant type system and apply it to a language using a front end style checker similar to Mungo. Padovani [25] extends their previous work using the *Join Calculus* as a basis for deadlock free typestate oriented programming. Their system can detect deadlocks from a composition of sub-systems that can each be checked in isolation. This paper focused on refining their previous work on modelling concurrent processes using their calculus. By defining dependency conditions and well-formedness for their type configurations to avoid deadlocks the author presents an interesting avenue for more work on typestate to model concurrent processes.

Papaya Jakobsen, Ravier, and Dardha presented *Papaya* [18] a typestate system that can perform global typestate analysis with unrestricted aliasing on objects. Their system allows unrestricted aliasing of objects by using a global type system which analyses the global program graph for object usage. The language they present is similar to Mungo in syntax but the typing system differs by using the global approach and focusing on building a program graph through method calls where at each method the protocol is checked to ensure the caller is in the correct state to make this call. This differs from the implementation found in Mungo which focuses on inferring

the usage of linear object and then checking it conforms to its typestate. The global type system allows changes to the object to be made by other methods that have an alias to this object, ensuring the typestate is still correctly followed. They then present Papaya as a plugin for Scala which follows their type system, allowing for typestate analysis of objects in the presence of unrestricted aliasing. The authors conclude that the runtime overhead of this system may be larger than that of the original Mungo system but suggest possible future work where the system can split up restricted and unrestricted objects based on if they need to be aliased.

Mungo and StMungo Kouzapas et al [20] presented StMungo and Mungo a static typestate toolchain based on session types. The authors base their typestate theory on multiparty session types, with the intuition being that session types capture the permitted sequence of communication calls between processes. They use this idea and translate it to capture the permitted sequence of method calls that an object can perform, following the typestate definition. The first tool the authors present is *StMungo*, this tool translates *Scribble* [31] local protocols into typestate protocols for Mungo, their second tool. Mungo extends Java with typestate definitions, these definitions attach a state machine to the class and dictate the correct sequence of methods an object can perform based on its state. The Mungo language requires linear usage of objects, so forbids aliasing. The type system for Mungo works by inferring a typestate definition for classes based on their usage and is then checked against the declared typestate. This inference system allows the programmer to ignore annotating method parameters and fields with typestates. They prove *Progress* and *Type Preservation* for their language and show an extended case study of typechecking an SMTP client. This paper followed previous work on Session types to bridge the gap between typestates and session types and provided a system that can typecheck the annotated Java program without requiring the creation of a new programming language. The practically of their toolchain is showcased with their presentation of a large case study. Recent work has improved upon Mungo in different areas including memory safety, and unrestricted aliasing.

Null Safe Mungo Mungo was extended by Bravetti et al [7] to provide null value checking to provide memory and method safety. They update the type system found in Mungo to check for null pointer dereferencing and memory leaks by ensuring classes complete their protocols. They use Mungo as their base to provide typestate aware analysis of null pointer errors, a novel contribution. Classes are analysed in isolation, which leads to a more complex system of protocol fidelty compared with the original Mungo paper. They show subject reduction and type safety. This paper presented an interesting and novel extension of the Mungo system and language, showing the possibilities of extending the system to type check different styles of errors.

Coping With Reality In Coping With Reality [23], Mota provides an update of the Mungo system with a large suite of features, as well as providing it in Kotlin instead of Java. Multiple features have been added to this system over Mungo including annotations that can be attached to field within typestate definitions of classes. Some of the new features include *droppable* states which allows the programmer to explicitly set some states as ”droppable” which allows the object to be terminated without error. The new system ensures that all non droppable states must transition to end states so ensures the correct and complete usage of a protocol.The user can also

annotate standard java libraries with typestates, while also writing *Stub files* which allows fields to be annotated without replacing the existing method implementations. This thesis provided a large suite of new features that Mungo could use while improving on some of the shortcomings of the original system.

4 Proposed Approach

Mungo as presented in the paper by Kouzapas et al [20] has been chosen as the typestate system to extend. This paper presents a core calculus which this use for their typestate inference system. The proposed approach is to update and extend this calculus with time constraints as presented by Timed Automata [2]. As the Mungo system is based on the theory of Multiparty Session Types [17], with a translation between *local session types* and *typestates* for classes. Due to this translation, Timed Multiparty Session Types [6] will be the main paper related to time to leverage during the extension of the core calculus. This paper provides in depth theory of updating multiparty session types with time constraints and includes proofs for time safety.

By updating the core calculus of Mungo by leveraging existing theory of timed multiparty session types I believe the extension will be feasible and achievable.

After updating the existing theory the Mungo system will be updated to include the time constraints following the core semantics. The system is currently implemented using the JastAdd framework [14], this implementation will be updated following the JastAdd documentation and newly updated semantics. This work will be carried out after proving the safety and progress properties of the system. To show the merit of this system a large scale case study will be carried out to implement a SMTP client, ensuring conformity to timed protocols.

5 Progress

Progress has been made on updating the existing core semantics of the Mungo system. Some figures of the updated theory have been excluded due to space constraints.

Top level syntax The core syntax and operational semantics of Mungo has been updated to include the notion of time and clock constraints as shown in Fig. 1. Typestate method signatures have been updated with optional time constraints following the theory presented in Timed Automata [2], this includes clock constraints and reset predicates. A new typestate been be introduced to distinguish between *declared* typestates and *inferred* typestates as shown in Fig. 2. Mungo uses an inference system where it infers a typestate that an object has used and then compares this inferred typestate to the declared typestate of the class. Inferred typestates have their syntax updated to be able to track time. Declared typestates have their syntax updated to include time constraints. A *delay* expression has also been added to allow a programmer to model the passing of time.

Type Decl	D	::=	$\texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \mid \texttt{enum}\ E\ \{\widetilde{l}\}$
Typestate	S	::=	$\widetilde{H} \mid \mu X.S \mid X$
Signature	H	::=	$[\kappa]\ T\ m(T) : S \mid E\ m(T) : \langle [\kappa_l]\ S_l \rangle_{l \in E}$
Field	F	::=	$T\ f$
Type	T	::=	$C \mid E \mid \texttt{bool} \mid \texttt{void}$
Method	M	::=	$T\ m(T\ x)\ \{e\}$
Constant	c	::=	$l \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{null} \mid *$
Path	r	::=	$\texttt{this} \mid r.f \mid x$
Expr	e	::=	$c \mid r \mid r.m(e) \mid r.f = e \mid e; e \mid r.f = \texttt{new}\ C \mid \lambda : e$
			$\mid \texttt{continue}\ \lambda \mid \texttt{switch}\ (e)\ \{e_l\}_{l \in E} \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e \mid \mathit{delay}(t)$
Time Constraint	κ	::=	(δ, P)
Clock Constraint	δ	::=	$\mathit{true} \mid \pi > c \mid \pi = c \mid \neg\delta \mid \delta_1 \wedge \delta_2$
Reset Predicate	P	::=	$\widetilde{\pi}$

Figure 1: Top-level syntax

Runtime syntax Runtime syntax has been updated to include a *delay* expression. This delay expression will model time passing in method expressions. This syntax is used to define the operation semantics of the core calculus.

Typestate inference system The inference system Mungo uses has been updated so inferred typestates can track details of time by annotating typestates with timing information based on the delay expressions. The current inference rules for expressions have been updated to include a rule that handles *Delay* expressions. Inference rules for methods and classes is used by Mungo to ensure a declared typestate is consistent. These rules have been updated to match the new typestate definitions.

Subtyping System After Mungo has inferred a typestate for an Object it uses a *subtyping* system to check that it is a valid sequence of methods calls under the declared typestate. In order to update the subtyping system to work with time. Inferred typestates carry the notion of the time of method calls while declared typestates contain time constraints on those method calls. The subtyping system will be updated to include relations and rules that ensure method calls from an inferred typestate only subtypes a declared typestate if it matches clock constraints present on the method. The subtyping rules can be seen in Fig. 3.

Transition Relations Transitions relations have been updated to reflect the new definitions for the declared and inferred typestates.

Reduction Relation A reduction relation has been presented for typing contexts by the authors of Mungo. The authors use this in order to prove progress and safety. This relation has had suitable updates to ensure these properties of the system can be proved with the new time updates.

$$
\begin{array}{llll}
\text{Inferred-Typestate} & I & ::= & \widetilde{B} \mid \mu X.I \mid X \\
\text{Inferred-Signature} & N & ::= & T\ m(T) : I \mid E\ m(T) : \langle I_l \rangle_{l \in E} \\
\text{Inferred-Branch} & B & ::= & N \mid (t, I)
\end{array}
$$

Figure 2: Inferred Typestate Syntax

$$
\text{S-Start}\ \frac{\emptyset \vdash I \leqslant_{\mathsf{sbt}} (v, S)}{I \leqslant_{\mathsf{sbt}} (v, S)} \qquad \text{S-End}\ \frac{}{\mathcal{R} \vdash \mathsf{end} \leqslant_{\mathsf{sbt}} \mathsf{end}} \qquad \text{S-Terminate}\ \frac{(I, (v, S)) \in \mathcal{R}}{\mathcal{R} \vdash I \leqslant_{\mathsf{sbt}} (v, S)}
$$

$$
\text{S-Method}\ \frac{\mathcal{R} \vdash I \leqslant_{\mathsf{sbt}} (v, S)}{\mathcal{R} \vdash T'\ m(T) : I \leqslant_{\mathsf{sbt}} (v,\ T'\ m(T) : S)} \qquad \text{S-Method-Time}\ \frac{v \models \delta \qquad v' = v[P \to 0] \qquad \mathcal{R} \vdash I \leqslant_{\mathsf{sbt}} (v', S)}{\mathcal{R} \vdash T'\ m(T) : I \leqslant_{\mathsf{sbt}} (v, (\delta, P)\ T'\ m(T) : S)}
$$

$$
\text{S-Rec1}\ \frac{\mathcal{R} \cup \{(I, (v, \mu X.S))\} \vdash I \leqslant_{\mathsf{sbt}} (v, S\{\mu X.S'/X\})}{\mathcal{R} \vdash I \leqslant_{\mathsf{sbt}} (v, \mu X.S)}
$$

$$
\text{S-Branches}\ \frac{\widetilde{B} \neq \emptyset \qquad \forall B \in \widetilde{B}, \exists H \in \widetilde{H}.\ \mathcal{R} \vdash B \leqslant_{\mathsf{sbt}} (v, H)}{\mathcal{R} \vdash \widetilde{B} \leqslant_{\mathsf{sbt}} (v, \widetilde{H})} \qquad \text{S-Time}\ \frac{\mathcal{R} \vdash I \leqslant_{\mathsf{sbt}} (v + t, S)}{\mathcal{R} \vdash (t, I) \leqslant_{\mathsf{sbt}} (v, S)}
$$

$$
\text{S-Bot}\ \frac{}{\mathtt{bot} \leqslant_{\mathsf{sbt}} U} \qquad \text{S-Grnd}\ \frac{U \in \{E, \mathtt{bool}, \mathtt{void}\}}{U \leqslant_{\mathsf{sbt}} U} \qquad \text{S-Empty}\ \frac{}{\emptyset \leqslant_{\mathsf{sbt}} \Delta}
$$

$$
\text{S-Delta}\ \frac{\Delta \leqslant_{\mathsf{sbt}} \Delta' \qquad U \leqslant_{\mathsf{sbt}} U'}{\Delta, r : U \leqslant_{\mathsf{sbt}} \Delta', r : U'} \qquad \text{S-Lambda}\ \frac{\Delta \leqslant_{\mathsf{sbt}} \Delta'}{\Delta, \lambda : X \leqslant_{\mathsf{sbt}} \Delta', \lambda : X}
$$

Figure 3: Subtyping relation (symmetric rule S-Rec2 omitted)

6 Work Plan

- **Week 12 (6/12)** Handwritten examples of typestate inference and subtyping. **Deliverable** - Examples written up in LaTeX
- **Week 13 (13/12)** Start of progress and safety proofs
- Winter Break
- **Week 14 (10/1)** Finish off proofs
- **Week 15 (17/1)** Read up on JastAdd framework and create development plan
- **Week 16 (24/1)** Being development of Mungo system
- **Week 17 (31/1)** Continue Mungo development. **Deliverable** - Correct Mungo implementation with time constraints

$$\textsc{Void}\ \frac{}{\Delta \vdash * : \texttt{void} \dashv \Delta} \qquad \textsc{Bool}\ \frac{}{\Delta \vdash \texttt{tt}, \texttt{ff} : \texttt{bool} \dashv \Delta} \qquad \textsc{Enum}\ \frac{l \in \mathsf{enums}(E)}{\Delta \vdash l : E \dashv \Delta} \qquad \textsc{Null}\ \frac{\texttt{class}\ C : S\ \{\widetilde{F}; \widetilde{M}\} \in \widetilde{D}}{\Delta \vdash \texttt{null} : C[\texttt{end}] \dashv \Delta}$$

$$\textsc{Weaken}\ \frac{\Delta \vdash e : U \dashv \Delta' \qquad r \notin dom(\Delta')}{\Delta \vdash e : U \dashv \Delta', r : C[\texttt{end}]} \qquad \textsc{Strengthen}\ \frac{\Delta, r : C[\texttt{end}] \vdash e : U \dashv \Delta'}{\Delta \vdash e : U \dashv \Delta'}$$

$$\textsc{PathC}\ \frac{U \neq C[I]}{\Delta, r : U \vdash r : U \dashv \Delta, r : U} \qquad \textsc{PathR}\ \frac{r \neq \texttt{this}}{\Delta, r : C[I] \vdash r : C[I] \dashv \Delta, r : C[\texttt{end}]}$$

$$\textsc{Equiv}\ \frac{\Delta \vdash e : U \dashv \Delta' \qquad \Delta =_{\mathsf{sbt}} \Delta''}{\Delta'' \vdash e : U \dashv \Delta'} \qquad \textsc{AsgnC}\ \frac{U \neq C[I] \quad \Delta \vdash e : U \dashv \Delta', r : U}{\Delta \vdash r = e : \texttt{void} \dashv \Delta', r : U}$$

$$\textsc{AsgnR}\ \frac{r \neq \texttt{this} \quad \Delta \vdash e : C[I] \dashv \Delta', r : C[\texttt{end}]}{\Delta \vdash r = e : \texttt{void} \dashv \Delta', r : C[I]}$$

$$\textsc{New}\ \frac{\begin{array}{c} clocks(S) = v \qquad r \neq \texttt{this} \qquad I \preccurlyeq_{\mathsf{sbt}} (init(v), S) \qquad \mathsf{typestate}(C) = S \\ \forall r.f : C'[I'] \in \Delta \implies I' = \texttt{end} \end{array}}{\Delta, r : C[\texttt{end}] \vdash r = \texttt{new}\ C : \texttt{void} \dashv \Delta, r : C[I]}$$

Figure 4: Typestate inference rules for expressions (part 1)

$$\textsc{Delay}\ \frac{\begin{array}{c} \Delta = \{r : C[(t, I)] \mid r : C[I] \in \Delta' \wedge I \neq \texttt{end}\} \cup \{r : U \mid r : U \in \Delta' \wedge U \neq C'[I']\} \\ \cup \{r : C''[end] \mid r : C''[end] \in \Delta'\} \end{array}}{\Delta \vdash delay(t) : \texttt{void} \dashv \Delta'}$$

$$\textsc{Seq}\ \frac{\begin{array}{c} \Delta \vdash e_1 : U' \dashv \Delta'' \\ \Delta'' \vdash e_2 : U \dashv \Delta' \\ U' \neq C[I] \qquad U' \neq \texttt{bot} \end{array}}{\Delta \vdash e_1; e_2 : U \dashv \Delta'} \qquad \textsc{Call}\ \frac{\begin{array}{c} T\ m(T'\ x)\ \{e'\} \in \mathsf{methods}(C) \quad I =_{\mathsf{sbt}} removeDelays(I') \\ \Delta'', r : C[I'], x : U' \vdash e'\{r/\texttt{this}\} : U \dashv \Delta', r : C[I], x : \mathsf{initT}(T') \\ \Delta \vdash e : U' \dashv \Delta'', r : C[\{T\ m(T') : I\}] \end{array}}{\Delta \vdash r.m(e) : U \dashv \Delta', r : C[I]}$$

$$\textsc{Switch}\ \frac{\begin{array}{c} \forall l \in E.\ \Delta_l, r : C[I_l] \vdash e_l : U_l \dashv \Delta' \\ \Delta \vdash r.m(e) : E \dashv \Delta'' \quad \Delta'' = \mathsf{join}(\{\Delta_l\}_{l \in E}) \end{array}}{\Delta \vdash \texttt{switch}\ (r.m(e))\ \{e_l\}_{l \in E} : \mathsf{join}(\{U_l\}_{l \in E}) \dashv \Delta'} \qquad \textsc{If}\ \frac{\begin{array}{c} \Delta_1 \vdash e_1 : U_1 \dashv \Delta' \quad \Delta_2 \vdash e_2 : U_2 \dashv \Delta' \\ \Delta'' = \mathsf{join}(\Delta_1, \Delta_2) \quad \Delta \vdash e : \texttt{bool} \dashv \Delta'' \end{array}}{\Delta \vdash \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 : \mathsf{join}(U_1, U_2) \dashv \Delta'}$$

$$\textsc{LExpr}\ \frac{\begin{array}{c} \Delta'' \vdash e : U \dashv \Delta', \lambda : X \qquad X \text{ fresh} \\ \Delta = \{r : C[\mu X.I] \mid r : C[I] \in \Delta''\} \cup \\ \{r : U' \mid r : U' \in \Delta'' \text{ and } U' \neq C'[I']\} \end{array}}{\Delta \vdash \lambda : e : U \dashv \Delta'} \qquad \textsc{Continue}\ \frac{\begin{array}{c} \Delta = \{r : C[X] \mid r : C[I] \in \Delta'\} \cup \\ \{r : U \mid r : U \in \Delta' \text{ and } U \neq C'[I']\} \end{array}}{\Delta \vdash \texttt{continue}\ \lambda : \texttt{bot} \dashv \Delta', \lambda : X}$$

Figure 5: Typestate inference rules for expressions (part 2)

- **Week 18 (7/2)** Begin creation of SMTP case study
- **Week 19 (14/2)** Continue case study, begin to create typestates and classes
- **Week 20 (21/2)** Investigate update of StMungo tool if time permitting
- **Week 21 (28/2)** Implement update of StMungo tool to convert timed Scribble protocols to new timed Mungo typestates. **Deliverable** - Updated StMungo tool.
- **Week 22 (7/3)** Implementation cleanup
- **Week 23 (14/3)** Start writing dissertation
- **Week 24 (21/3)** Continue dissertation writing
- **Week 25 (28/3)** Continue dissertation writing. **Deliverable** - First draft sent to supervisor
- **Week 26 (4/4)** Clean up dissertation
- **Week 27 (11/4)** Submit dissertation, deadline 17/4.

Week 20 states the possibility of updating the StMungo tool presented by kouzapas et al [20]. This tool converts session protocol written using Scribble [31] into typestate definitions Mungo can use. In their recent work Neykova et al updated Scribble with time and clock constraints. An interesting area of work if time is permitting would be to update the current StMungo tool to be able to convert these new timed Scribble protocol. This extra work is ambitious and relies on the plan above going perfectly with no major issues presenting itself in the implementation. This work will be left as future work if any issues arise.

References

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary 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, October 2009. Association for Computing Machinery.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

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

[4] Laura Bocchi, Julien Lange, and Nobuko 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.

[5] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous Timed Session Types: From Duality to Time-Sensitive Processes. In *Programming Languages and Systems*, volume 11423. Springer International Publishing, Cham.

[6] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In *CONCUR 2014 – Concurrency Theory*, volume 8704. Springer Berlin Heidelberg, Berlin, Heidelberg.

[7] Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias S. Jakobsen, Mikkel K. Kettunen, and António 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.

[8] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoretical Computer Science*, 410(2-3):142–167, February 2009.

[9] Silvia Crafa and Luca 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, October 2015. Association for Computing Machinery.

[10] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded Session Types for Object Oriented Languages. In *Formal Methods for Components and Objects*, volume 4709. Springer Berlin Heidelberg, Berlin, Heidelberg.

[11] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP 2006 – Object-Oriented Programming*, volume 4067. Springer Berlin Heidelberg, Berlin, Heidelberg.

[12] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A Distributed Object-Oriented Language with Session Types. In *Trustworthy Global Computing*, volume 3705. Springer Berlin Heidelberg, Berlin, Heidelberg.

[13] Simon J Gay, Vasco T Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z Caldeira. Modular session types for distributed object-oriented programming. *ACM Sigplan Notices*, 45(1):299–312, 2010.

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

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

[16] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, volume 1381. Springer Berlin Heidelberg, Berlin, Heidelberg.

[17] Kohei Honda, Nobuko Yoshida, and Marco 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, January 2008. Association for Computing Machinery.

[18] Mathias Jakobsen, Alice Ravier, and Ornela 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, September 2021. Association for Computing Machinery.

[19] Dr. John C. Klensin. Simple Mail Transfer Protocol. RFC 5321, October 2008.

[20] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming*, 155:52–75, April 2018.

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

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

[23] João Daniel da Luz Mota. Coping with the reality: adding crucial features to a typestate-oriented language. February 2021. Accepted: 2021-09-29T10:28:46Z.

[24] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, September 2017.

[25] Luca Padovani. Deadlock-Free Typestate-Oriented Programming. December 2017.

[26] MD Sharul. A visual formalism for complex systems. *Science of Computer Programming*, 8:10–1016, 1987.

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

[28] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. *ACM SIGPLAN Notices*, 46(10):713–732, October 2011.

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

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

[31] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In *Trustworthy Global Computing*, Lecture Notes in Computer Science, Cham. Springer International Publishing.