Towards Runtime Adaptation of Actor Systems

Ian Cassar

Supervisor: Dr. Adrian Francalanza

Faculty of ICT

University of Malta

29th May 2015

Submitted in partial fulfillment of the requirements for the degree of M.Sc. I.C.T. (Computer Science)
Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Towards Runtime Adaptation of Actor Systems

submitted is my work, except where acknowledged and referenced.

Ian Cassar

29th May 2015
Acknowledgements

I would honestly like to express my gratitude towards my supervisor Dr. Adrian Francalanza, whom despite being abroad and busy with his own research he always found enough time and patience to provide me with essential guidance and support. His excellent guidance, patience and tutoring made it possible for me to understand a lot of complex material that helped me make further progress in my work. I have learnt a lot from him and I must say that it has been quite a privilege working alongside him for the second time.

I would also like to thank my parents for always offering me their full support during my tough times. Even when I started to have doubts in my abilities, they never stopped believing in me, and always provided me with essential moral support. Moreover I would like to thank my loving girlfriend Melanie whom despite being busy with her final year in medical school, she was always by my side when I needed her most. The support of my loved ones kept me persevering and believing in myself even when I was facing tough situations. I therefore dedicate this dissertation to all of them.

Finally I would like to acknowledge that the research work disclosed in this dissertation was partially funded by the Master It! scholarship scheme (Malta).
“When there’s a will, there’s a way.”

– An Old English Proverb
The research work disclosed in this publication is partially funded by the Master it! Scholarship Scheme (Malta). This Scholarship is part-financed by the European Union – European Social Fund (ESF) under Operational Programme II – Cohesion Policy 2007-2013, “Empowering People for More Jobs and a Better Quality Of Life. Investing in your future
Abstract

Runtime Adaptation (RA) is a technique prevalent to long-running, highly available software systems, whereby system characteristics are altered dynamically in response to runtime events, while causing limited disruption to the execution of the system. Actor-based, concurrent systems are often used to build such long-running, reactive systems which require suffering from limited downtime. This makes runtime adaptation an appealing technique for mitigating erroneous behaviour in actor-systems, since mitigation is carried out while the system executes.

In this dissertation we focus on providing effective adaptations that can be localised and applied to specific concurrent actors, thereby only causing a temporary disruption to the parts of the system requiring mitigation, while leaving the rest of the system intact. We make the application of localised adaptations efficient through incremental synchronisation, whereby the specifier can strategically suspend specific parts of the system, whenever this is strictly required for ensuring that adaptations are effectively applied. We also study static analysis techniques to determine whether the specified incremental synchronisation is in some sense adequate for local adaptations to be carried out.

We thus identify a number of generic adaptations that can be applied to any actor system, regardless of its design and the code that it executes. We implement the identified adaptations as an extension of an existing Runtime Verification tool for actor-systems, thereby creating a RA framework for monitoring and mitigating actor systems. In parallel to our implementation we also develop a formal model of our RA framework that further serves to guide our implementation. This model also enables us to better understand the subtle errors that erroneously specified adaptation scripts may introduce. We thus develop a static type system for detecting and rejecting erroneous adaptation scripts prior to deployment, thereby providing the specifier with assistance for writing valid scripts. Although the static typesystem analyses scripts with respect to certain assumptions, we do not assume that the monitored system abides by these assumptions. We therefore augment our RA framework with dynamic checks for halting monitoring whenever the system deviates from our assumption. Based on this dynamically checked model of our RA framework, we prove type soundness for our static type system.

As a result of this dissertation we thus implement and formalise a novel Runtime Adaptation framework for actor systems as extension to an existing Runtime verification tool. Furthermore, exploring the mixture of static and dynamic typechecking, in the context of runtime verification, for the purpose of adaptation is also quite novel. This exploration lead to the developing a novel type analysis technique for detecting erroneously specified runtime adaptation scripts.
List of Figures

1.1 An example server actor implementation . . . . . . . . . . . . . . . . . 3
2.1 Runtime Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Runtime Adaptation . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 Runtime Enforcement . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.4 The Online monitoring Spectrum. . . . . . . . . . . . . . . . . . . . 16
2.5 Completely-synchronous Monitoring (CS) . . . . . . . . . . . . . . . . 17
2.6 Synchronous Monitoring with Synchronous Instrumentation (SMSI) . . 18
2.7 Asynchronous Monitoring with Checkpoints (AMC) . . . . . . . . . . . 19
2.8 Asynchronous Monitoring with Synchronous Detection (AMSD) . . . . 20
2.9 Completely-asynchronous Monitoring (CA) . . . . . . . . . . . . . . . . 21
2.10 The Logic and its Semantics . . . . . . . . . . . . . . . . . . . . . . . 28
3.1 A High-level illustration of Synchronous Instrumentation Monitoring. . 33
3.2 A High-level depiction of the Synchronous Event Monitoring protocol. . 36
3.3 A High-level illustration of Synchronous Detection (Hybrid) monitoring. 37
3.4 A High-level depiction of the Hybrid Monitoring Protocol. . . . . . . . 42
3.5 Yaws client connection protocol . . . . . . . . . . . . . . . . . . . . 43
3.6 Performance impact analysis (using unoptimised monitors) . . . . . . 49
3.7 Performance impact analysis (using statically optimised monitors) . . . 50
3.8 Performance impact analysis (using fully optimised monitors) . . . . . 51
4.1 Triggering the self-adaptive functionality of a system. . . . . . . . . . . 56
4.2 Applying adaptations using interrupts. . . . . . . . . . . . . . . . . . . 61
4.3 Applying adaptations encoded as messages. . . . . . . . . . . . . . . . 61
4.4 A conceptual protocol for applying synchronous adaptations. . . . . . 63
4.5 A high-level depiction of the Runtime Adaptation protocol. . . . . . . 68
4.6 The Logic and its LTS Semantics . . . . . . . . . . . . . . . . . . . . 75
4.7 A Runtime Semantics for Instrumented Properties with Adaptations . . 84
5.1 A Type System for Adaptation sHML scripts . . . . . . . . . . . . . . . 96
5.2 Dynamically Typed Adaptation-Script Rules . . . . . . . . . . . . . . 119
5.3 The Instrumentation Transition Rules . . . . . . . . . . . . . . . . . . . 120
A.1 Monitor Combinator Generation for Formula (A.1) . . . . . . . . . . . . 151
A.2 Optimised Synthesis for Formula (A.1) . . . . . . . . . . . . . . . . . 154
1. Introduction

Runtime Adaptation (RA) \cite{51, 50, 49} is a technique prevalent to long-running, highly available software systems, whereby system characteristics (e.g., its structure, locality etc.) are altered dynamically in response to runtime events (e.g., detected hardware faults or software bugs, changes in system loads), while causing limited disruption to the execution of the system. Numerous examples can be found in service-oriented architectures \cite{63, 48} (e.g., cloud-services, web-services, etc.) for self-configuring, self-optimising and self-healing purposes \cite{2}; the inherent component-based, decoupled organisation of such systems facilitates the implementation of adaptive actions affecting a subset of the system while allowing other parts to continue executing normally.

Actor systems \cite{4, 43, 17} consist of independently-executing components called actors. Every actor is uniquely-identifiable with a unique process id, has its own local memory, and can also spawn other actors and interact with them through asynchronous messaging. Actors systems are often used to build the aforementioned service-oriented systems with limited downtime \cite{7, 52, 43}. Actor-oriented coding practices such as fail-fast design-patterns \cite{17, 43} already advocate for a degree of RA that contribute towards building robust, fault-tolerant systems. Such systems are developed using mechanisms such as process linking and supervision trees, in which supervisor actors can detect crashed actors and respond through adaptations such as restarting the actors, reverting them through a previous (or different) implementation version, or else killing further actors that may potentially be affected by the crash.
1. Introduction

This dissertation studies ways how runtime adaptation for actor systems can be extended to be able to respond to sets of runtime events that go beyond actor crashes. More specifically, we would like to observe sequences of events that allow us to invoke an adaptation action to mitigate the effects of the detected misbehaviour. This technique, however, can also be used to observe positive (liveness) events that allow us to adapt the system to execute more efficiently (e.g., by switching off unused parts). More generally, we intend to develop a framework for extending actor-system functionality through RA, so as to improve aspects such as resilience and resource management.

Due to the inherent concurrency of actor-systems, adaptation actions can be achieved using generic, actor-level manipulations (e.g., restart misbehaving actors) to mitigate only the concurrent components (actors) that contributed to the detected misbehaviour, thus leaving the other actors unaffected. Although similar adaptations are already being employed by using certain system design conventions (e.g., fail-fast, supervision trees, etc.), it can be quite challenging to implement effective adaptations that can be applied on any system without requiring it to be developed in a particular manner. This is because effective adaptations require the monitor to temporarily synchronise with the components in need of adaptation, so as to achieve a tighter level of control upon them. This allows for the adaptations to apply the required mitigation in a timely manner thereby increasing the effectiveness of the applied mitigation. However, actor systems do not natively support synchronisation. Implementing such an RA framework is even more challenging as to our knowledge, runtime adaptation wrt. actor systems has not yet been explored, and is therefore quite novel.

1.1 Aims and Objectives

We aim to introduce Runtime Adaptation for actor systems by extending existing Runtime Verification (RV) tools, such as [68, 37, 23, 39], with RA functionality. The appeal of such an approach is that RV tools already provide mechanisms for specifying the system behaviour needed to be observed, together with instrumentation mechanisms for
1. Introduction

Figure 1.1: An example server actor implementation offering integer increment and decrement services.

observing such behaviour. We also aim to introduce adaptations that can be localised to individual components, thereby allowing for specific actors to be mitigated while leaving the other actors unaffected.

As a proof-of-concept, we aim to focus on one of these actor-based RV tools called detectEr [39] — an RV tool for monitoring invariant (safety) properties about long-running (reactive) actor-based systems written in Erlang [7]; presently this tool is only able to perform detections that are exclusively asynchronous. Using this tool we want to investigate ways how to insert a degree of synchrony so as to allow for detections to be replaced by effective adaptation actions that respond to behaviours detected, while reusing as many elements as possible from the existing technology.

Example 1.1. In Fig. 1.1 we present a simple running example system to which we refer in different parts throughout this dissertation. This figure depicts an actor-based server consisting of:

- a front-end common-interface actor with identifier \( i \), receiving client requests;
- a back-end incrementor actor with identifier \( j \), handling integer increment requests;
- and a back-end decrementor actor \( k \), handling decrement requests.

A client sends service requests to actor \( i \) of the form \( \{ \text{tag}, \text{arg}, \text{ret} \} \) where: \( \text{tag} \) selects the type of service, \( \text{arg} \) carries the service arguments and \( \text{ret} \) specifies the return address for the result (typically the client actor ID). The interface actor forwards the request to one of its back-end servers (depending on the \( \text{tag} \)) whereas the back-end servers process the requests, sending results (or error messages) to \( \text{ret} \). With respect to this simple system we can define several safety (invariant) RV properties including Property [7].
Property 1. For every increment request, \(\{\text{inc}, n, \text{cli}\}\), that is received by the common-interface \(i\), the requesting client, \(\text{cli}\), should never be sent an error message \(\text{err}\).

With an RA framework developed as an extension of an RV setup, RV properties can easily be augmented with adaptation actions that the monitor can apply upon detecting a violation. For instance, RV \(\text{property1}\) can thus be extended into RA \(\text{property2}\) below.

Property 2. For every increment request, \(\{\text{inc}, n, \text{cli}\}\), that is received by the common-interface \(i\), should the requesting client, \(\text{cli}\), be sent an error message \(\text{err}\), this should be mitigated by: restarting the common-interface \(i\), and purging the mailbox of the system actor that sent the error message to the client.

To fulfill our aim of developing such a runtime adaptation framework for actor systems, we divide our work into the following three objectives:

(i) As a minor objective, we first want to carry out a preliminary study \(\text{wrt.}\) the different techniques by which synchronous monitoring can be introduced in asynchronous actor systems, and thus integrate them within the detectEr RV tool. Although it is generally accepted that synchrony increases monitoring overheads, we are not aware of any studies that attempt to quantify and assess by how much, especially \(\text{wrt.}\) actor systems. Hence we also want to carry out an impact assessment based on the studied synchronous monitoring techniques (\(\text{i.e.}\), the ones we integrate in detectEr). This would aid us in identifying the ideal synchronisation mechanism for introducing effective adaptation actions within detectEr in an efficient way.

(ii) More importantly, we want to identify and implement generic adaptation actions that are relevant in the context actor systems. To be able to effectively apply these adaptations we want to build upon the efficient synchronous monitoring candidate technique identified in (i), so as to permit the monitor to temporarily achieve tighter control over the system (or parts of it) thereby being able to effectively execute the required mitigation. Using this synchronisation technique and the identified adaptations, we want to extend detectEr along with its specification language resulting
1. Introduction

in an effective RA framework for actor systems. This framework should allow the specifier to apply the necessary adaptations on any actor-based system (written in Erlang) regardless of the code it executes or the way it is designed and implemented. Providing the monitor with more control can however lead to introducing errors in the system. To be able to better identify and study these errors we want to develop a formal model wrt. the implementation constraints of our RA framework.

(iii) Formalising our RA framework puts us in a position to identify and study in more depth the errors that our monitors may introduce, without having to deal with the complexities of the implementation. We conjecture that some of the identified errors can be statically detected by thoroughly inspecting the adaptation scripts prior to deployment. We therefore want to look into static analysis techniques that can assist the specifier into writing error-free adaptation scripts.

Each objective contributes in its own way towards developing a more refined Runtime Adaptation framework for actor systems. For instance in (i) we address performance aspects related to developing an efficient (yet effective) RA framework; while in (ii) we identify the relevant adaptation actions and address their implementability issues. This contributes to having a more effective RA framework that provides the specifier with an adequate number of generic adaptations that can be applied to any actor-based (Erlang) system. Furthermore, formalisation also contributes towards developing a more understandable RA framework. In fact in (iii) we plan to exploit the understandability of the formal model to address certain incorrectness that the RA monitors may now introduce. This contributes towards a more conducive RA framework which provides guidance to the specifier for writing error-free adaptation scripts.

1.2 Document Outline

In this thesis we address our three objective in Chapter3,4 and 5 Each of these chapters presents and evaluates the contributions required for fulfilling the respective objective. More specifically, we structure our document in the following manner:
1. Introduction

- In Chapter 2, we provide the reader with the necessary background material for better understanding our work. We define and differentiate between Runtime Verification, Adaptation and Enforcement. We also identify a spectrum of online monitoring approaches ranging from complete synchronous and complete asynchronous monitoring on opposite ends of the spectrum. Finally, we present the actor model as implemented by a host language called Erlang. This is followed by an overview of the specification language used by our target RV tool, detectEr, for specifying RV properties for Erlang systems.

- In Chapter 3, we explore the design space for synchronisation mechanisms and assess the respective overheads induced as a result of increasing synchrony when monitoring actor systems. Based on the results of this impact assessment, we identify an efficient synchronisation technique that guides the implementation of our RA adaptation framework presented in Chapter 4.

- In Chapter 4, we identify relevant adaptations and build on the efficient synchronisation technique identified in Chapter 3. This allows for the implementation of more effective adaptations which require the monitor to achieve tighter control over the system. We then formalise the behaviour of our runtime adaptation scripts as operational semantics. This allows for better understanding the core concepts of our runtime adaptation framework without having to deal with the complexities of the implementation.

- As erroneous adaptation scripts can introduce errors, in Chapter 5, we present a static type system for identifying potential problems that may arise from erroneous scripts, prior to deployment. Finally, we evaluate the type system by proving type soundness for typed monitors.

- Finally, in Chapter 6, we present other work related to different aspects of our research. We then conclude this document and provide future work suggestions in Chapter 7.

- The appendix chapters A, B and C respectively provide: information about detectEr’s
1. Introduction

monitor optimisations; additional RV properties used in our impact assessment along with detailed result tables; and the proofs for auxiliary lemmas required when proving theorems introduced in Chapters 4 and 5.
2. Background

In this chapter we provide the reader with the necessary background information for better understanding our work. More importantly we aim to disambiguate terms and techniques that may sometimes be confused for one another, while we also want to establish what we truly mean when we refer to certain terms during the course of this dissertation.

For instance, in Sec. 2.1 we define and disambiguate between three monitoring techniques, namely Runtime Verification, Runtime Adaptation and Runtime Enforcement. This is intended to clarify what we mean by these three terms, as the distinction between them is not always made clear in the current literature.

Similarly, in Sec. 2.2 we disambiguate between two different classes of monitoring techniques found in the current literature, namely Offline and Online monitors; whereby we expand on the latter by looking into different definitions of online monitoring. Based on these definitions we devise a spectrum of online monitoring techniques wrt. component-based systems. Furthermore, we also define our own terms to identify and distinguish between the techniques in the spectrum. This is intended to help the reader understand the design space that we need to explore for augmenting the detectEr RV tool with synchrony in Chapter 3.

Following this, in Sec. 2.3 we present an overview of the actor model with respect to Erlang – a programming language that implements this model. This is intended to aid the reader in understanding the core concepts of actor-systems, along with technical
(implementation-oriented) issues regarding the Erlang programming language.

Finally, in Sec. 2.4 we present a detailed overview of the detectEr RV tool, particularly about its specification logic. This is aimed to aid the reader in understanding the syntax and semantics of the logic used by this tool, upon which we build and extend in this dissertation.

2.1 Runtime Verification, Adaptation and Enforcement

Ensuring formal correctness for actor-based, concurrent systems is a difficult task, primarily because exhaustive, static analysis verification techniques such as model-checking quickly run into state-explosion problems. This is typically caused by multiple thread interleavings of the system being analysed, and by the range of data the system can input and react to.

We therefore look into three different (yet related) dynamic monitoring techniques that overcome the issues of static analysis techniques by verifying only a single execution, normally the currently executing one. In literature the boundaries between these techniques are often blurred; we therefore want to disambiguate between these monitoring techniques and establish upfront what we mean by the terms Runtime Verification (RV), Runtime Adaptation (RA) and Runtime Enforcement (RE).

The first technique that we consider is Runtime verification (RV) [10, 55]. This is a lightweight dynamic verification technique that provides an appealing compromise towards ensuring a degree of formal correctness without running into such scalability issues. These issues are avoid by only verifying that a specific execution of a system (usually the currently executing one) satisfies or violates a specific correctness property.

In most runtime verification settings [10, 38, 39, 33] a correctness property $\varphi$ is generally specified in terms of a logic with precise formal semantics, and then automatically synthesised into an executable monitor $M(\varphi)$. This monitor is essentially the executable implementation of the abstract property, which actually checks whether a given execution trace of a specific system, violates (or satisfies) the property it was derived from. As
2. Background

illustrated in Fig. 2.1 the monitors in runtime verification usually have a passive role \cite{10,39}. In fact, they are generally concerned exclusively with receiving system events, analysing them and detecting (flagging) violations (or satisfactions) of their respective correctness properties. Hence RV monitors refrain from directly modifying the system’s behaviour in any way.

![Figure 2.1: Runtime Verification](image)

Monitors in Runtime Adaptation \cite{63,51,49,67} break the passivity of RV monitors. In fact, RA monitors are generally provided with more control over the system, enabling them to automatically execute adaptation actions after analysing a particular sequence of system events. For instance as shown in Fig. 2.2 rather than flagging violations, RA monitors can therefore execute adaptation actions upon detecting an event sequence that denotes incorrect behaviour. In this way they aim to mitigate and reduce the effects of the detected violation. The adaptation actions executed by the monitor, do not necessarily correct or revert the detected misbehaviour \cite{59,51}; instead they change certain aspects of the system as it executes, with the aim of preventing either future occurrences of the same error, or of other errors that may potentially occur as a side-effect of the detected violation.

However, the use of runtime adaptation goes beyond self-healing \cite{2} i.e., detecting and mitigating misbehaviour. In fact RA can also be used to optimise \cite{2,51} the system’s behaviour based on the information collected by the monitor, e.g., switch off redundant processes when under a small load, or increase processes and load balancing when under a heavy load. These type of adaptations contribute towards making the monitored system
2. Background

Runtime Enforcement (RE) \[36, 57, 56\] is the strictest monitoring technique, whereby correct behaviour is kept in line by anticipating incorrect behaviour and countering it before it actually happens. In fact as depicted in Fig. 2.3 in runtime enforcement techniques the monitor and the system are so tightly coupled to the extent that the monitor acts as a proxy which wraps around the system and analyses its external interactions (see the dotted-line in Fig. 2.3). In this way the monitor is thus able to either drop incorrect events by preventing certain system actions from occurring; or else add system events by executing actions on behalf of the system, that should have been executed by the system but where not \[57, 56\].

The distinction between RV, RA and RE is however often blurred in the current liter-
2. Background

ature. For instance, we find a number of RV tools such as [25, 24, 23, 18] that allow for user defined actions to be executed whenever certain conditions are met. This distinction is further blurred as Runtime Adaptation may also be used to achieve Runtime Enforcement by anticipating system misbehaviour and executing stricter and more corrective adaptations. This may require adaptations to be tailor-made for the respective system, to totally prevent incorrect behaviour from manifesting. However, unlike RE, RA typically allows for violations to occur, but then executes remedial actions to mitigate the effects [51] of the committed violations. During the course of this thesis we focus on Runtime Adaptation as defined in Fig. 2.2.

2.2 Synchronous and Asynchronous monitoring

As developing a runtime adaptation framework on top of a runtime verification setup requires introducing a degree of synchrony in our monitoring, we look into the different types of monitoring techniques found in the current literature. Furthermore, as monitors are usually derived from high-level properties, the same property may be converted into different types of monitors which may apply different instrumentation techniques for checking whether the property was satisfied or not.

In Sec. 2.2.1 we therefore conjecture that there are two main classes of monitoring approaches, namely, Online and Offline. Particularly, we look into the latter class (i.e., offline) as this is often ambiguated with asynchronous monitoring, whereas for the former class (i.e., online) in Sec. 2.2.2 we discuss the different definitions that one can find in the current RV literature. Based on these definitions, in Sec. 2.2.3 we present a spectrum of online monitoring approaches wrt. to component-based systems, that vary in the level of coupling and control that they posses over the system’s components.

2.2.1 Offline and Online monitoring

We divide runtime monitoring into the following two main classes: (i) Online monitoring and (ii) Offline monitoring. Although our work deals exclusively with online monitoring,
2. Background

We opt to define offline monitoring such that we are able to distinguish it from other monitoring techniques.

**Online Monitoring**

In an online monitoring setup [55, 10, 31, 44], an executing system is dynamically monitored for violations (or satisfactions) *during the course of its execution*. Online monitors are therefore developed in such a way that they execute alongside the system by verifying its execution in an incremental fashion [55, 10]. This means that the monitor must be able to receive notifications about relevant events occurring in the executing system and make a decision based on the current information collected so far.

As online monitors are developed to verify currently executing systems, they are also capable of making early detections. This is sometimes exploited by RV tools (such as in [25, 24]) to allow for the execution of user defined actions that may potentially be used to mitigate the corruption suffered by the system as a result of a property violation.

The primary disadvantage of online monitoring is that it imposes an inevitable runtime overhead on the system given that additional monitoring code is added to the system. As runtime overheads are a very undesirable side effect of online monitors, a good deal of effort is usually devoted to create highly efficient monitors that keep this overhead as low as possible.

**Offline Monitoring**

Unlike online monitoring, in offline monitoring [55, 10, 19, 35, 61] the system is not directly monitored as it executes, instead the relevant system events are recorded as an execution trace inside a data store. Once the monitored system terminates (or whenever a satisfactory number of events have been recorded) the collected execution trace is forwarded to the offline monitor. The offline monitor, which is entirely independent from the system, then proceeds by inspecting the system events recorded in the trace, and given that the trace provides enough information, it deduces whether the property it represents was satisfied or violated.
2. Background

Offline monitors are particularly suitable for properties that can only be verified by globally analysing the complete execution trace which is generated once the system stops executing. In fact some properties may require to be globally analysed using a backward traversal [61, 66] of the trace. This monitoring mode is less intrusive than online monitoring, as it does not interfere with the system except for the logging of events, thus imposing much less overheads. This, however, comes at the cost of late detections, given that violations are detected either after the system stops or else after the system is done executing a considerably large amount of actions which are recorded and then inspected by the monitor.

2.2.2 Varying definitions of Online Monitoring approaches

In the current literature we often encounter different definitions of online monitoring which usually vary in two aspects: (i) the level of coupling between the monitor and the system, and (ii) the level of control that the monitor has over the system.

The definition of synchronous online monitoring presented in [33, 31, 9], requires the system to work in lockstep with the monitoring code, whereby the entire monitored system is bound to block and wait until the necessary monitoring code is done handling an event generated by a system component. These type of online monitors are therefore very tightly coupled to the system and have a very tight level of control over the system to the extent that they cause all system components to block even when only a single component generates an event. As this extreme synchronous monitoring technique uses a high amount of synchronisation is generally appealing for inherently synchronous systems [31, 9] such as circuits and embedded systems.

The work presented in [19, 21], implement a less stringent definition of synchronous online monitoring in which the monitors are only allowed to block the concurrent component which generates the event. In this way other concurrent components may carry on with their normal execution, even while the component which generated the event is interrupted until the required monitoring is carried out. Although these monitors still posses a relatively tight level of coupling and control over the system, they are able to
2. Background

provide timely detections without resorting to a high use of synchronisation as opposed to the definitions presented in [33, 31, 9].

Yet another online monitoring approach is proposed in [27, 22], allowing the user to manually specify synchronisation checkpoints where the system (or certain parts of it) should temporarily stop and wait for the monitor to finish handling monitoring requests. In this way the system is able to asynchronously send monitoring events to the monitor, without waiting for the monitoring code to complete. Explicit checkpoints can then be employed to temporarily block certain parts of the system thereby increasing the monitor’s level of control over the system which allows the lagging monitor to catch up with the system execution. The use of checkpoints are especially useful when monitoring for the occurrence of certain safety-critical events, in which late detection is not feasible. Checkpoints therefore permit the user to optimize the synthesised monitors by manually determining the level coupling and control that the monitor possesses over the system.

In [66] synchronous monitoring is defined as a monitoring approach which provides timely detections. This definition describes an even less stringent monitoring approach in which the monitors are loosely-coupled from the system yet retain enough control over the system so that they can provide timely detections. We conjecture that implementing synchronous monitoring \( \text{wrt.} \) this definition requires monitoring most of the system events asynchronously, yet utilise synchronisation at strategic points which might lead to a violation. In this way violations are still detected in a timely manner.

Finally, in this thesis we follow the definitions for asynchronous online monitoring given in [27, 66, 22, 23]. These definitions state that although the synthesised monitors still execute alongside the system, they are however \textit{loosely-coupled} from the system to the extent that they have barely any control over the monitored system and may suffer from \textit{late detections}. In fact whenever a specified event is performed by the system, an event notification is asynchronously forwarded to the monitor without suspending any part of the system. The monitor is then able to verify the received event notifications at its own pace independently from the system. Importantly note that unlike offline monitors (defined in Sec. 2.2.1), asynchronous monitors still execute alongside the system.
and still have to analyse the systems trace of events in an incremental fashion. Due to this lack of synchrony and control, asynchronous monitors are presumed to impose considerably lower overheads compared to other approaches. However, most of the time an asynchronous monitor is unable to detect violations immediately and carry out effective mitigation actions.

2.2.3 The Spectrum of Online Monitoring Approaches

Based on the range of definitions presented in Sec. 2.2.2, we devise a spectrum of online monitoring approaches which we explain wrt. component-based systems, whereby components represent concurrent entities (e.g., node, thread, actor, . . .) that can be in either two states, namely blocked or running. As depicted in Figure 2.4, this spectrum ranges from a tightly coupled completely-synchronous monitoring approach on one end, to a loosely-coupled completely-asynchronous monitoring approach on the other end.

The spectrum also presents currently known online monitoring approaches that lie in between these two extremes. These middle way approaches provide a tradeoff between the level of coupling and control that the monitor has upon the system. It is generally assumed that monitoring approaches that are closer to the synchronous end of the spectrum tend to posses a higher level of control over the system, which comes at the expense of higher the overheads. Conversely, approaches which are closer to the opposite end are generally presumed to be more efficient. However, so far we are not aware of any publications that conducted a thorough investigation to confirm (or reject) this general assumption.

|       | CS | SMSI | AMC | AMSD | CA      |
|-------|----|------|-----|------|---------|
| Tightly Coupled |    |      |     |      |         |
|       |    |      |     |      | Loosely Coupled |

Figure 2.4: The Online monitoring Spectrum.
2. Background

**Completely-synchronous Monitoring (CS)**

As depicted in Fig. 2.5, **completely-synchronous monitoring** \[33, 31, 9\] refers to the synchronous extreme of the spectrum. This implies that the system and the monitors are extremely *tightly coupled* together to the extent that whenever an event occurs in *one* component of the system, such as C1, the *entire* system execution is *interrupted* (as shown by (1) in Fig. 2.5). The system components remain blocked until the necessary monitoring checks are performed, in which case the monitor unblocks the components (as shown by (2)).

![Figure 2.5: Completely-synchronous Monitoring (CS)](image)

Since completely-synchronous monitoring generally provides immediate detections, it can therefore be used to effectively mitigate incorrect behaviour. This approach is however highly intrusive as it introduces an unnecessarily high level of synchronisation that might lead to an infeasible performance degradation on the monitored system. Completely-synchronous monitoring is generally applied on small synchronous systems \[31, 9\] such as embedded systems and circuits.

**Synchronous Monitoring with Synchronous Instrumentation (SMSI)**

*Synchronous Monitoring with Synchronous Instrumentation* \[21, 19\] is a monitoring approach which is closer to a completely-synchronous approach yet is less intrusive. In fact as shown in Fig. 2.6, this approach assumes that whenever a *single component* of
the monitored system, such as $C_1$, executes a specified event, then only the execution of this component is interrupted (as shown by (1) in Fig. 2.6). This component remains blocked until monitoring completes, in which case the component is reset to a running state (as shown by (2)).

![Figure 2.6: Synchronous Monitoring with Synchronous Instrumentation (SMSI)](image)

Hence synchronous instrumentation monitoring reduces the level of overall synchronisation by relinquishing control over the other components, thereby allowing them to keep executing whenever a certain component generates an event. However, this monitoring technique still possesses a relatively high control over the system as it still needs to synchronously inspect each and every event, even though it only interrupts the normal execution of the component which generated the event until monitoring completes.

**Asynchronous Monitoring with Checkpoints (AMC)**

This approach [27, 22, 33] allows for an asynchronous decoupling between the system (or parts of it) and monitor executions yet also provides the user with the ability to specify checkpoints where the decoupled system and monitor executions should synchronise. We conjecture that in terms of component-based systems, checkpoints may also be associated with specific system components. For instance as shown by (1) and (2) in Fig. 2.7 an RV user may use a checkpoint specify that certain system components such as $C_1$ and $C_2$ should temporarily block their execution and synchronise. Such synchronisation can
be done during periods where the system is not required to be responsive, and therefore affords to wait for the monitor to catch-up with the system execution.

![Diagram](image)

**Figure 2.7: Asynchronous Monitoring with Checkpoints (AMC)**

As shown by (3), once the monitor synchronises with the system, it unblocks the respective system components and allows them to continue generating further event notifications. An appealing feature of this monitoring approach is that it enables the user to manually determine the level of synchrony and control that the monitor can possess over the system. The user may also use this mechanism to specify that the system should stop and wait for the monitor’s analysis to complete when certain safety-critical events occur during execution [27]. This ensures that safety-critical violations are detected in a timely fashion thus permitting the monitor to effectively react to the violation and possibly invoke mitigating actions. To further reduce the level of monitor intrusion, the user may specify that the system should keep on executing whenever it generates non-safety critical events.
Asynchronous Monitoring with Synchronous Detection (AMSD)

Asynchronous Monitoring with synchronous detection (based on the definition given in [66]) is yet another monitoring approach that lies closely to completely-asynchronous monitoring in our spectrum. This approach uses minimal synchronisation to ensure synchronous (timely) detections, by synchronising only for system events that might directly contribute to a violation. In Fig. 2.8 we illustrate this concept wrt. an example that assumes a simple transaction system which must satisfy the following invariant property:

**Property 3.** A user cannot make a transaction before a login.

Property 3 is thus only violated whenever a transaction event (trans) is perceived without a preceding login event. This means that to timely detect violations of property 3 it suffices that the monitor synchronises only with a system component whenever this performs a transaction event. As shown by (1) in Fig. 2.8 only component C2 is blocked, as this produced a transaction event that may lead to a property violation. By contrast C1 kept on executing even though it produced a login event. In (2) we see that the monitor...
managed to detect that C2 violated property as it did not login prior to performing the transaction. The detection was made in a timely fashion as C2 was not allowed (in (1)) to execute any further after committing the transaction event that lead to the violation. Furthermore in (2) one can notice that C1 blocked waiting for the monitor’s verdict as a result of producing a transaction event, in which case the monitor detects that C1 had already performed a login event meaning that it did not violate property. Hence in (3) the monitor allows C1 to proceed.

**Completely-asynchronous Monitoring (CA)**

Finally, in *Completely-asynchronous monitoring* [27, 39, 22, 23] the monitors are designed to be as loosely coupled as possible from the system they are monitoring. In fact completely-asynchronous monitors are designed to listen for system events and handle them in the “background” without interfering in any way with the system execution. Hence, as shown in Fig. 2.9 in completely-asynchronous monitoring, the system is allowed to proceed immediately after placing an event notification (e.g., C1 event 1) in the monitor’s buffer. The monitor can then independently read the event notifications from its buffer and carry out the necessary checks at its own pace.

![Figure 2.9: Completely-asynchronous Monitoring (CA)](image)

Although this approach imposes minimal intrusion over the monitored system, it suffers from late detection which makes it highly-unlikely for an asynchronous monitor to be able to effectively mitigate the detected misbehaviour. As offline monitoring [33, 66], is inherently asynchronous, it is sometimes classified as being an asynchronous monitoring approach. However, we conjecture that offline monitoring generally requires
2. Background

prerecorded (generally complete) traces, while CA monitoring can incrementally analyse partial traces that can be extended as the system proceeds with its execution.

Conclusion

We have introduced the concept of a spectrum of online monitoring approaches, and explained each approach wrt. component based systems. The spectrum consists in:

- **Completely-synchronous monitoring** on one end — this achieves an extremely high level of control over the system that could potentially deter monitoring efficiency; and

- **Completely-asynchronous monitoring** on the opposite end — this is presumed to provide efficient monitors with very low intrusion over the monitored system, by sacrificing the level of control over the monitored system, including timely detections.

We have also identified a number of middle-way approaches that lie in between these two online monitoring extremes in our spectrum. We also conjecture that there may be other middle-way approaches apart from the ones that we identified which still need to be explored. The identified approaches however allow us to better understand the design space that we need to explore and assess in Chapter 3 in order to achieve our first objective (see objective (i) in Sec. 1.1), i.e., that of increasing the level of synchrony in the detectEr RV tool while preserving monitoring efficiency.

2.3 The Actor-Model and Erlang

The component based systems that we consider in this thesis are actor-based [5] systems written in a programming language called Erlang [7, 58]. In Erlang processes (i.e., actors) are threads of execution that are uniquely identified by a process identifier and posses their own local memory. Erlang processes execute asynchronously to one another, interacting through asynchronous messages instead of sharing data.
In fact, processes can only communicate by explicitly sending a copy of their data to the destination process (using the unique identifier as address). These messages are received at a process mailbox (a form of message buffer) and can be exclusively read at a later stage by the process owning the mailbox. Since asynchronous messages may reach a mailbox in a different order than the one intended, the (mailbox) read construct uses pattern matching to allow a process to retrieve the first message in the mailbox matching a pattern, possibly reading messages out of order. Whenever a retrieved pattern is not matched, the executing process blocks, waiting for a message matching this pattern to reach its mailbox. Apart from the mailbox, actors possess their own private state known as the process dictionary, in which they can associate data entries with a unique key, which can then be used in different points during the actor’s execution to retrieve or modify the stored data entry.

Processes may also be registered under a specific name. This allows other actors to send messages to a registered process by using its name rather than its process id. This is useful to avoid having to communicate the process id of an (important) actor, to the other actors. Furthermore, processes may spawn other processes at runtime, and communicate locally-known (i.e., private) process identifiers as messages. A Process may not only use the known process identifiers to communicate messages, but also to forcibly terminate or link processes \cite{58}. Implementation wise, Erlang processes are relatively lightweight \cite{7}, and language coding practices \cite{58} recommend the use of concurrent processes wherever possible. These factors ultimately lead to systems made up of independently-executing components (i.e., processes) that are more scalable, maintainable, and use the underlying parallel architectures more efficiently \cite{17}.

At its core, Erlang is dynamically-typed, and function calls with mismatching parameters are typically detected at runtime. This implies that Erlang provides very few static guarantees thereby making it prone to more dynamic errors; this makes Runtime Adaptation an appealing approach for detecting and mitigating dynamic errors. Function definitions are named and organised in uniquely-named modules. Within a particular module, there can be more than one function with the same name, as long as these
names have different arities (i.e., number of parameters). Thus, in Erlang, every function is uniquely identified by the triple `module_name:function_name/arity` e.g., `math:mul/2` identifies a function named `mul` which takes two arguments and is defined in the `math` module.

Erlang also offers a number of fault-tolerance mechanisms including process linking [7, 58], whereby whenever a process fails i.e., terminates abnormally, all the other actors linked to it are terminated as well. However, Erlang also provides a trapping mechanism whereby rather then failing, the linked processes are sent a message in their mailbox notifying them that a linked process has failed. Furthermore, linking may be either bi-directional or uni-directional [7]. This means that if an actor A links with B using a bi-directional link, then should either one of the linked actors crash, then the other actor crashes as well (or is notified in case of trapping). Whereas if A links with B using a uni-directional link, then only A gets terminated (or notified) in case the linked actor B crashes, i.e., if A crashes, B remains unaffected.

Erlang systems are often structured by the supervisor pattern which is built on linking and trapping, in which processes at the system fringes are encouraged to fail-fast when an error occurs (as opposed to handling the error locally). In this way the linked supervisor processes [17] is kept responsible to carry out the necessary adaptations for detecting the crash and handling the error. However, misbehaviour does not necessarily lead to crashing an actor, for instance if an invalid computation is saved to a database, then none of the actors are likely to crash. Therefore, a more refined adaptation mechanism built over a runtime verification setup provides a good alternative for detecting and mitigating such misbehaviour.

As actors in Erlang are inherently processes that interact using asynchronous message passing, they can be formally represented as Labelled Transition Systems (LTS) [70, 40, 14, 39] where at an abstract level, computation can be viewed as a graph of states connected by labelled arcs denoting the communicated messages.
2. Background

2.4 detectEr – A concurrent RV tool

In [39], the authors present detectEr – a provably correct, actor-based runtime verification tool written in Erlang. This tool uses a syntactic subset of the modal μ-calculus called Safe Hennessy-Milner logic (sHML) [3] to specify safety properties over Labelled Transition System (LTS). As Erlang programs can be formalised in terms of an LTS, sHML can therefore be used to specify properties for Erlang systems as well.

Moreover, the monitors synthesised by this tool, employ a completely asynchronous (CA) monitoring approach for detecting the runtime violations of the respective formulas. This implies that the synthesised monitors barely have any control over the monitored system. Asynchronous monitoring is in fact achieved using the tracing mechanism offered by the EVM OTP platform libraries [58] which do not require instrumentation at source-code level. Instead VM directives generate trace messages for specified execution events that are sent to a specially-designated tracer actor i.e., to the synthesised monitor.

Hence to introduce synchrony in detectEr it is likely that we need to completely redesign the mechanism for forwarding trace messages to the monitor, such that the system and the monitor are allowed to synchronise whenever a trace event is exchanged.

2.4.1 The Syntax

The tool’s sHML syntax [39] [3] is defined inductively using the BNF in Fig. 2.10. It is parametrised by a set of boolean expressions, \( b, c \in \text{Bool} \), equipped with a decidable evaluation function, \( b \Downarrow v \) where \( v \in \{\text{true, false}\} \), and a set of system actions \( \alpha, \beta \in \text{Act} \). It assumes two distinct denumerable sets of variables:

- **Term variables**, \( x, y, \ldots \in \text{VAR} \) — used in action patterns to quantify over data values, and to evaluate boolean conditions defined in if-statements; and

- **Formula variables**, \( X, Y, \ldots \in \text{LVAR} \) — used to define recursive (logical) formulas.

Note that even though we work up-to \( \alpha \)-conversion (for both variable sets), detectEr accordingly renames duplicate variables during a pre-processing phase.
2. Background

Formulas $\varphi, \psi \in \text{ Frm}$ include: truth tt, and falsity ff, conjunctions, $\varphi \& \psi$, modal necessities, $[e]\varphi$ for specifying particular actions that the system may perform, maximal fixpoints to describe recursive properties, $\max X. \varphi$, and if-statements to analyse system data, if $b$ then $\varphi$ else $\psi$. A necessity formula, $[e]\varphi$ specifies a pattern $e$ denoting the following system actions:

- **message input**: e.g., $i?3$ – process $i$ must receive the value 3;
- **message output**: e.g., $i\triangleright j!3$ – process $i$ must send $j$ the value 3;
- **function call**: e.g., $i\triangleright \text{call(mod, fun, [1, test])}$ – process $i$ must call a function $\text{fun}$, which is defined in module $\text{mod}$, using arguments $[1, \text{test}]$; and
- **function return**: e.g., $i\triangleright \text{ret(mod, fun, 2, \{res, 5\})}$ – process $i$ must return from function $\text{fun}$ (which takes 2 arguments) with return value $\{\text{res, 5}\}$.

**Remark 1.** Although from a theoretical standpoint sHML specifications should limit themselves to input and output actions so as to treat individual components as blackboxes, monitoring for function calls and returns is essential. This is because certain output and input actions in actual Erlang implementations may be abstracted away inside the function calls of system libraries, making them (directly) unobservable to the instrumentation mechanism. However, these actions (and the data associated with them) can still be observed indirectly, through the calls and returns of the functions that abstract them.

A pattern $e$ in a necessity $[e]\varphi$ may also contain term variables that pattern-match with actual (closed) system actions, thus acting as a binder for these variables in the continuation subformula $\varphi$; to improve readability, we sometimes denote term variables that are not used in the continuation subformula as underscores, _. Similarly a recursive definition $\max X. \varphi$, serves as a binder for recursive variable $X$ in $\varphi$.

**Example 2.4.1.** Recall Ex. 1.1.1 in which we define safety Property[7] for monitoring invariant behaviour in the actor-based server illustrated in Fig. 1.1.
2. Background

syntax we can now formally represent this property as the following formula:

\[ \varphi \overset{\text{def}}{=} \max Y. \left[ i?\{\text{inc}, x, y\} \left( (j\triangleright y!\{\text{res}, x + 1\}) Y \right) \& \left( j\triangleright y!\text{err}\right) \right] \] (2.1)

The tool detectEr allows us to specify the recursive safety property (2.1) (above) as follows: it requires that, from an external viewpoint, every increment request received by \( i \), i.e., action \( i?\{\text{inc}, x, y\} \), is followed by a service answer from \( j \) to the client address bound to variable \( y \), with value \( x + 1 \) as specified by action \( j\triangleright y!\{\text{res}, x + 1\} \) (and then recurs through variable \( Y \)). However, increment requests followed by an error message sent from any actor back to \( y \) (as defined by action \( j\triangleright y!\text{err} \)), represent a violation (i.e., \( \text{ff} \)). The detectEr compiler is then used to automatically synthesise a concurrent actor-based monitor corresponding to property (2.1). It also adds directives to Erlang VM to forward system execution events to the synthesised monitor [39].

Ex. 2.4.1 is just a pathological example showing that sHML is ideal for specifying infinite computation (using maximal fixpoints) in the form of invariant, safety properties such as formula (2.1). In the case of actor-based systems, this essential as the actor model is usually used to develop non-terminating, reactive systems [4, 43, 7].

2.4.2 The Model

The semantics [39] for sHML, given in Fig. 2.10, is defined over a formal Labelled Transition System (LTS) representation of an arbitrary Erlang program in the style of the operational models (for Erlang programs) presented in [70, 40, 14, 39]. In these operational models, the runtime semantics for Erlang programs is formally represented in the form of a set of actors \( A, B \in \text{Actr} \) composed in parallel. These actors can reduce either by using internal (unobservable) \( \tau \)-actions, \( A \xrightarrow{\tau} B \), or else by using external observable \( \alpha \) actions, \( A \xrightarrow{\alpha} B \), i.e., actions that an external actor such as the monitor can perceive. Weak transitions [39] for modeled Erlang systems i.e., \( A \xrightarrow{\text{weak}} (A \xrightarrow{\alpha} B) \), are often represented as \( A \xrightarrow{\alpha} B \).
BNF Syntax.
\[ \varphi, \psi \in \text{FRM} ::= \text{tt} \mid \text{ff} \mid \varphi \& \psi \mid [e] \varphi \mid X \mid \max X. \varphi \mid \text{if } b \text{ then } \varphi \text{ else } \psi \]

Satisfaction Semantics.
- \((A, \text{tt}) \in R\) always
- \((A, \text{ff}) \in R\) never
- \((A, \text{if } b \text{ then } \varphi \text{ else } \psi) \in R\) if \((b \iff \text{true} \land (A, \varphi) \in R) \lor (b \iff \text{false} \land (A, \varphi) \in R)\)
- \((A, \varphi \& \psi) \in R\) if \((A, \varphi) \in R \land (A, \psi) \in R\)
- \((A, [e] \varphi) \in R\) if \(A \overset{\alpha}{\Rightarrow} B \land \text{mtch}(e, \alpha) = \sigma\) then \((B, \varphi\sigma) \in R\)
- \((A, \max X. \varphi) \in R\) if \((A, \varphi[(\max X. \varphi)/X]) \in R\)

Violation Semantics.
- \(A, t \models_v \text{tt}\) never
- \(A, t \models_v \text{ff}\) always
- \(A, t \models_v \text{if } b \text{ then } \varphi \text{ else } \psi\) if \((b \iff \text{true} \land A, t \models_v \varphi) \lor (b \iff \text{false} \land A, t \models_v \psi)\)
- \(A, t \models_v \varphi \& \psi\) if \(A, t \models_v \varphi \lor A, t \models_v \psi\)
- \(A, t \models_v [e] \varphi\) if \(A \overset{\alpha}{\Rightarrow} B \land \text{mtch}(e, \alpha) = \sigma\) and \(B, t \models_v \varphi\sigma\)
- \(A, t \models_v \max X. \varphi\) if \(A, t \models_v [\max X. \varphi]/X\)

2.4.3 The Semantics

In [39] the authors define satisfaction semantics for the sHML logic over arbitrary Erlang programs modelled as an LTS (as discussed in Sec. 2.4.2), to denote how an sHML formula \(\varphi\) is satisfied by an Erlang program \(A\).

The satisfaction semantics state that a **satisfaction relation** \(R \in (\text{ACTR} \times \text{FRM})\) is **always** satisfied by \(\text{tt}\) but **never** satisfied by \(\text{ff}\), and in the case of conjunction formulas, \((\varphi \& \psi)\), both branches must be satisfied. Conditionals, if \(b \text{ then } \varphi \text{ else } \psi\), are only satisfied if the taken branch is satisfied. Modal necessities, \([e] \varphi\), are satisfied in two ways: either **trivially i.e.,** when necessity pattern \(e\) does not pattern match action \(\alpha\); or else since the resulting actors \(B\) satisfy \(\varphi\sigma\) after performing a pattern-matchable event action \(\alpha\) that yields substitution environment \(\sigma :: \text{VAR} \rightarrow \text{VAL}\).

Finally recursive definitions, \(\max X. \varphi\), are satisfied by a set of actors \(A\), if the unfolded formula \(\varphi[(\max X. \varphi)/X]\) (where every occurrence of formula variable \(X\) in \(\varphi\) is substituted by \(\max X. \varphi\)) is also satisfied by \(A\). This implies that recursive formulas may
potentially be satisfied by \textit{infinite computation}.

In \cite{39} the authors also argue that even though sHML can specify \textit{infinite behaviour}, the \textit{execution traces} that \textit{violate} safety properties (expressible through sHML) are always \textit{finite}. This implies that since runtime monitors can only work with finite traces \cite{10, 55}, they are therefore able to detect violations from finite violating traces. Based on this fact the authors formulate the violation semantics (presented in Fig. 2.10) and prove that these correspond to their satisfaction semantics.

The violation relation, denoted as \(\models_v\), is the least relation of the form (\(\text{Actr} \times \text{Act}^* \times \text{ Frm}\)) satisfying the rules defined in Fig. 2.10 (follows from the semantics given in \cite{39}). Regardless of the contents of a trace \(t\), all actor systems \(A\) can \textit{never} violate \(\mathbb{t}\), yet \textit{always} violate \(\mathbb{f}\). Conjunction formulas, \(\varphi \& \psi\), cause a violation if either \(\varphi\) or \(\psi\) lead to a violation. Conditionals, \(\text{if } b \text{ then } \varphi \text{ else } \psi\), only cause a violation in case the taken branch (\(\varphi\) or \(\psi\)) causes a violation. All actor systems \(A\) violate a necessity formulas \([e]\varphi\) whenever a pattern-matchable action \(a\) is performed (yielding substitution \(\sigma : \text{Var} \rightarrow \text{Val}\)), and the resulting actors \(B\) that are transitioned violate \(\varphi\sigma\). Note that actors that \textit{do not} perform any pattern-matchable actions trivially satisfy \([e]\varphi\), and hence cannot lead to a violation. A recursive formula \(\text{max } X. \varphi\) is violated whenever the unfolded formula \(\varphi[(\text{max } X. \varphi)/X]\) is also violated.

The violation relation is only satisfied by \textit{finite} traces, as opposed to the satisfaction relation which can also be satisfied by \textit{infinite} traces. Hence the violation semantics makes it easier for us to understand how the synthesised runtime monitors flag a violation \textit{wrt.} a finite trace. In this dissertation we therefore focus on the violation semantics.

\section{Conclusion}

We have provided the reader with the necessary background information for better understanding our work. We have also established what the terms Runtime Verification, Runtime Adaptation and Runtime Enforcement refer to in the context of this dissertation, given that the distinction between these techniques is not always made clear in the
current literature.

We have also clarified how we distinguish between Offline and Online monitors, and expanded on the latter by looking into different definitions of online monitoring, upon which we presented a spectrum of online monitoring techniques \textit{wrt.} component-based systems. Furthermore, we also define our own terms to identify and distinguish between the techniques in the spectrum. By identifying this spectrum we are now in a better position to understand the design space that we need to explore for introducing synchrony within the \texttt{detectEr} RV tool in Chapter 3.

Finally we provided an overview of the actor model as implemented by a host language Erlang, followed by a detailed overview of our target RV tool, \texttt{detectEr}, on which we build our runtime adaptation framework for actor systems in Chapter 4.
3. Assessing the Impact of Increasing Monitor Synchronisation

Effective runtime adaptation requires achieving a tighter level of control over the system so as to provide timely detections of violations, thereby allowing the monitor to mitigate erroneous actors before these are allowed to proceed. In this chapter we therefore conduct preliminary investigations for implementing synchronous monitoring mechanisms over an inherently asynchronous architecture (i.e., the actor model).

We conduct this investigation by exploring the design space of monitoring mechanisms identified in Sec. 2.2.3 and assess the repercussions that a monitored actor system may suffer due to synchrony. Although one might easily infer that synchronous monitoring incurs higher overheads as opposed to its asynchronous counterpart, we set out to investigate and quantify the overhead increase in an inherently asynchronous setting such as ours. This is important as in the case of asynchronous platforms where synchrony is not natively supported (such as the actor-model), introducing synchronous monitoring in a naive way might highly affect the performance of the monitored asynchronous system. This opposes inherently synchronous settings [31, 9] in which the impact of using synchronous monitoring might not be as significant. Quantifying the impact of synchronous monitoring on asynchronous actor-systems will then guide our design decisions for implementing adequate adaptation mechanisms in Chapter 4.

We start our investigations by first introducing tight, coarse-grained synchronisation...
3. Assessing the Impact of Increasing Monitor Synchronisation

in Sec. 3.1 as this is more straightforward to understand and implement. Following this in Sec. 3.2 we explore techniques for reducing the level of synchrony to a more fine-grained level, while still being able to achieve timely detections. Since synchronous communication is not native to asynchronous platforms (such as the actor model), introducing synchrony requires instrumenting the system to encode a synchronisation protocol that forces the system to wait until the monitor is done verifying a reported event. Encoding such a protocol may therefore introduce overheads that must be minimised as much as possible.

Hence in Sec. 3.3 we finally assess and quantify the monitoring overheads when using coarse-grained synchrony, fine-grained synchrony and asynchronous monitoring. We conduct our experiments wrt. Yaws – a third-party, actor-based webserver written in Erlang, by monitoring it for several safety properties expressed in detectEr’s logic, using these three different monitoring techniques.

3.1 Introducing Synchronous Monitoring

There are a number of ways how one can layer synchronous monitoring atop of an inherently asynchronous computational model, such as the actor model. For instance, one can insert actual monitoring functionality within the sequential thread of execution of each actor (in the style of [68, 19]) and then have the monitoring code (scattered across independently executing actors) synchronise, as required, in a choreographed setup [68, 38]. Instead, we opt for an orchestrated solution as shown in Fig. 3.1 (below).

In this setup, monitored actors are instrumented to synchronously report monitored actions as event notifications to a (conceptually) centralised monitoring setup that receives all reported events and performs the necessary checking. Note that although conceptually centralised, the orchestrator monitor is not monolithic [39] as it consists of independent, concurrent sub-monitors that execute independent of one another.

Furthermore, it is imperative that such a synchronous event reporting mechanism must prevent a system actor from progressing until the monitor is done checking its syn-
3. Assessing the Impact of Increasing Monitor Synchronisation

Figure 3.1: A High-level illustration of Synchronous Instrumentation Monitoring.

Chronously reported event. For instance in Fig. 3.1, system actor A must only be allowed to progress up to event A:evt2 after the monitor completes checking the previously reported event, i.e., A:evt1. However, while A is waiting for the monitor to finish verifying its reported event (i.e., A:evt1), the other concurrent actors B and C are still allowed to progress independently until they synchronously report their own events.

There are various reasons for opting for such an orchestrated setup. The main reason being that it allows us to perform a like-with-like comparison with the existing asynchronous setup present in detectEr, thus obtaining a more precise comparison between the relative overheads of synchronous and asynchronous monitoring. Furthermore, this setup conforms to the ideology of the actor oriented paradigm [7, 29], which strongly encourages the developer to divide large tasks into smaller subtasks that can execute concurrently whenever this is possible. This is therefore reflected in this setup, as the monitor and the system are kept as two completely separate entities. Also, by using this approach the instrumentation code at the system side is kept minimal, thus leaving the instrumented code close to the original. Moreover, monitoring is consolidated into a group of concurrent actors that are separate from the monitored system, thereby improving manageability (e.g., parts of the system may crash leaving the monitor unaffected).
3. Assessing the Impact of Increasing Monitor Synchronisation

3.1.1 Instrumentation through Aspect-Oriented Weaving

We consider Aspect Oriented Programming (AOP) as an ideal candidate for implementing synchronous event reporting. The instrumentation is carried out using an AOP Framework for Erlang [54, 41] (in the style of standard AOP tools such as [28]) to add the necessary instrumentation in the system code.

As detectEr supports the monitoring of output and input actions (i.e., message sends and receives) along with function calls and returns, we needed an AOP framework that supports instrumenting all of these actions. The AOP presented in [54] was not ideal as this only supports function calls and returns; however, this was later augmented with aspects for instrumenting the required input and output actions as a preliminary investigation in [41]. However, this AOP framework [41] required further refinements, which we conducted during the course of this research, as it was still unstable and too coarse-grained.

The extended aspect-based instrumentation requires an aspect file that specifies the actions requiring instrumentation, along with a purpose built module called advices.erl containing three types of advices used by the AOP injections, namely before_advice, after_advice and upon_advice advices. For the function call events specified in the aspect file, the AOP weaves before_advice advices reporting the event data to the monitor before the function is invoked. For message sends and function returns, the AOP weaves after_advice advices to report the committed action to the monitor once the action is performed. In the case of message sends, this allows for the system to actually send the required message prior to reporting the event to the monitor. In the case of function returns, after_advice advices are required since return values are only known after the function actually returns. The weaving concerning mailbox reading events (performed using the receive construct) was not as straightforward. As the Erlang receive construct may contain multiple pattern-matching guarded clauses i.e.,

```
  receive guard_1 -> expression_1 ; ... ; guard_n -> expression_n end.
```

The aspects for instrumenting input and output actions did not support for selectively instrumenting specific send and receive operations which match a specific pattern. This was therefore adding unnecessary instrumentation.

---

1The aspects for instrumenting input and output actions did not support for selectively instrumenting specific send and receive operations which match a specific pattern. This was therefore adding unnecessary instrumentation.
3. Assessing the Impact of Increasing Monitor Synchronisation

The AOP weaves upon_advice advice for each guarded expression that matches the message pattern defined by the receive aspects, as specified in the aspect file. For instance if we assume that only \( \text{guard}_1 \) and \( \text{guard}_2 \) match the receive patterns specified in the aspect file, we get

\[
\text{receive} \ \text{guard}_1 \rightarrow \text{upon_advice}(..), \text{expression}_1 ; \\
\text{guard}_2 \rightarrow \text{upon_advice}(..), \text{expression}_2 ; \\
\text{guard}_3 \rightarrow \text{expression}_3 ; \ldots ; \\
\text{guard}_n \rightarrow \text{expression}_n \end{equation}

as opposed to the coarse-grained instrumentation in [41] which adds upon_advice to every guarded expression thereby introducing unnecessary instrumentation. Also note that at runtime only one receive guarded expression is triggered. In the case if the triggered expression is an instrumented one, the necessary pattern-matched data of the event is extracted and thus can be reported to the monitor by the advice as required.

3.1.2 The Synchronous Instrumentation Protocol

Fig. 3.2 depicts a global implementation overview of the synchronous instrumentation monitoring (SMSI) protocol for two monitored events, i.e., events \( e_1 \) and \( e_2 \). This protocol denotes the messages exchanged between the instrumentation code injected in the asynchronous actor system and the monitor in order to achieve a synchronous event reporting mechanism. In fact this mechanism is achieved through handshakes over asynchronous messages between the two parties; this technique is a common practice for encoding synchronous message passing on asynchronous models such as the actor model [7].

This protocol starts with monitoring loop sending an initial acknowledgement message to the system, signalling it to execute until it produces the next monitored event. To synchronously report a monitored event, the instrumentation code at the system side extracts the necessary data associated with the event, sends it as a trace message to the monitor, and then pauses by issuing a receive statement that waits for an unpause-acknowledgement message from the monitor. Since, the (acknowledgement)
3. Assessing the Impact of Increasing Monitor Synchronisation

Asynchronous messages may get reordered in transit (potentially interfering with this protocol) the instrumentation code at the system side generates a unique nonce for every monitored event and sends it along with the event data. In return, when the monitor sends back the acknowledgement message, it includes this unique nonce. This allows the instrumentation code to pattern-match for (and possibly read out-of-order) mailbox inputs containing this nonce, and unblock based on this corresponding acknowledgement.

![Diagram](image)

Figure 3.2: A High-level depiction of the Synchronous Event Monitoring protocol.

The monitoring loop outlined in Fig. 3.2 corresponds to the pattern-matching functionality required by a necessity formula \([e] \varphi\). For instance, if the pattern is not matched (i.e., the necessity is trivially satisfied), it acknowledges immediately to the system to continue executing and terminates monitoring by issuing a `send_ack(Nonce)` command to release the blocked system actor. However, if the pattern matches there is still a chance that a violation can be detected and therefore proceeds by handling the event. Event handling involves executing internal (non-necessity) actions specified after the modal necessity (i.e., corresponding to \(\varphi\) in \([e] \varphi\)) such as: unfolding recursive definitions, evaluating if-statement conditions, spawning conjunction submonitors, etc. Crucially, if \(\varphi\) reduces into `ff` (e.g., after evaluating an if-statement), the monitor refrains from sending back the acknowledgement, thus blocking the offending system indefinitely while flagging the violation. In a runtime adaptation setup, this allows for effectively executing adaptation actions instead of flagging the violation. Otherwise, the monitor issues
command $\text{loop}(\text{Nonce}_2)$ to reiterate and release the blocked actor using the delegated nonce ($i.e.$, $\text{Nonce}_2$).

### 3.2 A Hybrid Instrumentation for Timely Detections

Since synchronous instrumentation monitoring (SMSI) is too coarse-grained (as it synchronises for every system event), we therefore devise an alternative fine-grained synchronisation strategy which achieves timely detections with less synchronisation. The main idea is to move away from synchronous instrumentation monitoring, towards using incremental synchronisation to obtain synchronous detection monitoring (AMSD) for actor systems. This is possible as in order to attain timely detections, the instrumentation need not require certain system components to execute in lockstep with the monitor for every monitored event leading to the violation. Instead, (expensive) synchronous event monitoring can be limited to critical actions, ($i.e.$, system actions contributing directly towards a violation), thereby letting the system execute in a decoupled fashion otherwise.

![Figure 3.3: A High-level illustration of Synchronous Detection (Hybrid) monitoring.](image)

For instance in Fig. 3.3 if we assume that events $A:evt3$ and $B:evt2$ are the only
system events that may lead to a violation, we can limit synchronous event monitoring to these two events only. Note that for now we make a simplifying assumption that for the logic used by detectEr (given in Fig. 2.10) these critical actions are necessity actions preceeding (directly or indirectly) a ff formula as explained in Ex. 3.2.1 below.

**Example 3.2.1.** Recall Property[7] which we formalised as formula (2.1) in Ex. 2.4.1 (restated below):

$$\phi \overset{df}{=} \max Y. [i?\{\text{inc}, x, y\}] ( [j>y\{\text{res}, x+1\}] Y \& ([_{\text{err}}y\text{ff}] ) ) \quad (2.1)$$

Based on our simplifying assumption, to synchronously detect a violation in this property, only action \([_{\text{err}}y\text{ff}]\) needs to be synchronously monitored, as it precedes a ff formula. The other actions (i.e., \([i?\{\text{inc}, x, y\}]\) and \([j>y\{\text{res}, x+1\}]\) can however be monitored for asynchronously, without affecting the timeliness of detections. ■

**Remark 2.** With this simplifying assumption we assume that the actor that commits the last action preceeding a violation (ff) is solely responsible for the detected violation, however this is not always the case.

For instance in the example system given in Fig. 1.1 despite the incrementor being the one to send the error message \text{err} to the connected client thus violating (2.1), this error was caused by an invalid interaction made by the common-interface in some previous step. Hence strictly speaking to truly achieve a timely detection, the common-interface should also be forbidden from progressing after performing an interaction that potentially contributes to the violation, until the monitor performs the necessary checks.

Our assumption however permits us to get a clear indication of the associated performance impacts without introducing too much complexities at this stage. Remember that for now our main objective is to identify an efficient synchronous monitoring mechanism upon which we can build in order to introduce runtime adaptation in Chapter 4. ■
3.2.1 Logic Extensions

To achieve synchronous detection monitoring we explore a hybrid technique that is capable of switching between synchronous and asynchronous event monitoring, and which limits synchronous monitoring to critical system actions. To achieve this we extend the syntax of detectEr’s logic [39] (outlined in Sec. 2.4) by two constructs: a synchronous false formula and a synchronous necessity formula, with violation semantics analogous to that of ff and $[e] \varphi$ respectively.

$$\varphi, \psi \in \text{Frm} ::= \ldots | \text{sff} \quad \text{(synchronous false)} | \text{[e] } \varphi \quad \text{(synchronous necessity)}$$

In the extended logic, formulas carry additional instrumentation information relating to how they need to be runtime-monitored. By default, all the monitoring is asynchronous, unless one specifies that a violation is to be synchronously detected, sff, or that a particular system event needs to be synchronously monitored, $[e] \varphi$. Although synchronous necessities can be used in specification scripts to engineer synchronisation checkpoints (i.e., to encode AMC; see Sec. 2.2.3) similar to what is presented in [26], in this study we limit their use to achieving timely-detections. In fact, rather then using synchronous necessities explicitly we encode synchronous falsities, sff, in terms of synchronous necessities as explained in Ex. 3.2.2.

Example 3.2.2. Recall the system presented in Fig. 1.1 for which we extend and refine formula (2.1) (restated in Ex. 3.2.1) as (3.1) (shown below). We now distinguish between two kinds of violations in which: result values $w$ that are less than $x + 1$ are defined to be asynchronously detected (i.e., if $w < x + 1$ then ff), while values greater than $x + 1$ are specified to be synchronously detected (i.e., if $w > x + 1$ then sff). We also specify that cases where client $y$ is sent an error (i.e., $\triangledown y!\text{err}$) are to be synchronously detected as well.

$$\max \ Y. [i?\{\text{inc}, x, y\}][([j\triangledown y!\{\text{res}, w\} (\text{if } w = x + 1 \text{ then } Y \\
& \text{& if } w > x + 1 \text{ then } \text{sff} \\
& \text{& if } w < x + 1 \text{ then } \text{ff} ) ) \ & ([\triangledown y!\text{err}] \text{sff} ) $$

(3.1)
3. Assessing the Impact of Increasing Monitor Synchronisation

The new monitor synthesis algorithm requires a pre-processing phase to determine which events are to be synchronously monitored in order to implement a synchronous fail. For instance, formulas \( [\vdash y ! err]sff \) and \( [\vdash y ! err]ff \) are both monitored in the same way, in fact the pre-processing function \( \text{syn} \text{enc} \), presented in Def. 3.2.1, encodes the former into the latter. As shown by Def. 3.2.1, generally determining the critical actions to synchronously monitor for implementing a synchronous fail is not as straightforward, since the first necessity formula preceding a \( sff \) may be interposed by intermediate formulas such as conjunctions and if-statements (as in the case of formula (3.1)).

**Def. 3.2.1.** \( \text{syn} \text{enc} : \text{Frm} \mapsto (\text{Bool} \times \text{Frm}) \)

\[
\text{syn} \text{enc}(\varphi) \equiv \begin{cases} 
    \langle \text{false}, [e]\psi' \rangle & \text{if } \varphi = [e]\psi \land \text{syn} \text{enc}(\psi) = \langle \text{true}, \psi' \rangle \\
    \langle \text{false}, [e]\psi' \rangle & \text{if } \varphi = [e]\psi \land \text{syn} \text{enc}(\psi) = \langle \text{false}, \psi' \rangle \\
    \langle b_1 \lor b_2, \psi_1' \land \psi_2' \rangle & \text{if } \varphi = \psi_1 \land \psi_2 \land \text{syn} \text{enc}(\psi_1) = \langle b_1, \psi_1' \rangle \\
    & \land \text{syn} \text{enc}(\psi_2) = \langle b_2, \psi_2' \rangle \\
    \langle b_1 \lor b_2, \text{if } c \text{ then } \psi_1' \text{ else } \psi_2' \rangle & \text{if } \varphi = \text{if } b \text{ then } \psi_1 \text{ else } \psi_2 \\
    & \land \text{syn} \text{enc}(\psi_1) = \langle b_1, \psi_1' \rangle \\
    & \land \text{syn} \text{enc}(\psi_2) = \langle b_2, \psi_2' \rangle \\
    \langle b, \max X. \psi' \rangle & \text{if } \varphi = \max X. \psi \land \text{syn} \text{enc}(\psi) = \langle b, \psi' \rangle \\
    \langle \text{true}, ff \rangle & \text{if } \varphi = sff \\
    \langle \text{false}, \varphi \rangle & \text{otherwise}
\end{cases}
\]

The pre-processing function \( \text{syn} \text{enc} \) recursively inspects a formula \( \varphi \) and returns a more implementable version in which the necessities specifying critical actions are identified and converted into synchronous necessities. For cases where \( \varphi \) reduces to \( sff \), this function returns a pair containing boolean value \( true \) and \( ff \). The boolean value \( true \) indicates that a synchronous falsity \( sff \) was found and converted to \( ff \), which is more implementable. This truth value then allows for the first necessity action declared prior to \( sff \) to be converted into a synchronous necessity. This guarantees that the falsity is detected on-time. In fact when inspecting necessity actions, \( [e]\psi \), the function recursively inspects the subsequent formula \( \psi \) (i.e., it reapplys itself as \( \text{syn} \text{enc}(\psi) \)) and if the recursive application returns the pair \( \langle true, \psi' \rangle \), it converts the necessity \( [e]\psi \) into a synchronous necessity \([e]\psi' \). This is possible as the truth value \( true \) in the returned pair, signifies that at least one \( sff \) was found somewhere along \( \psi \) and converted into \( ff \), thus
changing $\psi$ into $\psi'$ which is more implementable. After encoding a normal necessity into a synchronous necessity, the `syn_enc` function then returns a pair containing the encoded formula, i.e., $[[e]]\psi'$, and truth value `false`, where the latter denotes that the synchronous falsity has already been encoded. This prevents prior asynchronous necessities from being unnecessarily converted into synchronous ones.

To conduct our experiments we therefore augment the `syn_enc` function as a preprocessing phase in our compiler to encode formulas containing synchronous falsities into a more implementable versions as shown in Ex. 3.2.3.

**Example 3.2.3.** In the case of property (3.1) the compiler uses the `syn_enc` function to check whether there exists at least one path that leads directly to a synchronous fail. Since action $[\psi\downarrow y!\{\text{res, } w\}]$ is followed by a conjunction of three if-statements in which one of them leads to a synchronous falsity, i.e., if $w > x + 1$ then `sff`, then this necessity is converted into a synchronous necessity. More obviously action $[\nabla y!\text{err}]$ is also converted as it is immediately followed by `sff`. The compiler therefore converts property (3.1) into (3.2):

$$
\max Y. [i?\{\text{inc, } x, y\}] \left( [[\psi\downarrow y!\{\text{res, } w\}]] \left( \begin{array}{c} \text{if } w = x + 1 \text{ then } Y \\
\& \text{if } w > x + 1 \text{ then } \text{ff} \\
\& \text{if } w < x + 1 \text{ then } \text{ff} \\
\end{array} \right) \right) \& \left( [[\nabla y!\text{err}]] \text{ff} \right)
$$

(3.2) ■

### 3.2.2 The Hybrid Monitoring Protocol

In hybrid monitoring, both synchronous and asynchronous event monitoring require code instrumentation, as shown in Fig. 3.4 (below). *Asynchronous* necessity actions inject advice functions that send a monitoring message, e.g., event $e1$, to the monitor containing: the event details, e.g., $d1$, and a *null* nonce; this message is reported *without blocking* the system. Upon receiving the null nonce, the monitor determines that it does not need to send an acknowledgment back to the system. Furthermore, *Synchronous* necessity actions are implemented as per synchronous instrumentation (see Sec. 3.2), where the system attaches a *fresh* nonce along with the event data to the monitoring message thereby notifying the monitor that it requires to be acknowledged back.
3. Assessing the Impact of Increasing Monitor Synchronisation

3.3 Evaluation

By now the extended version of the detectEr RV tool supports three methods for monitoring actor-based systems, namely:

- **Asynchronous Monitoring (CA)**, which was the only monitoring mechanism originally supported by detectEr;

- **Synchronous Instrumentation Monitoring (SMSI)**, introduced in Sec. 3.1;

- **Hybrid Monitoring (AMSD)**, introduced in Sec. 3.2.

Using these three monitoring approaches, we want to carry out experiments in which we monitor for several safety properties written specifically for the Yaws webserver. With these experiments we aim to obtain performance results for each approach in order to be able to compare and contrast the performance overheads imposed by these monitoring techniques. The results of these experiments will then serve as guidance for choosing the appropriate instrumentation mechanism on which we can build our RA framework in Chapter 4.

---

Figure 3.4: A High-level depiction of the Hybrid Monitoring Protocol.
3. Assessing the Impact of Increasing Monitor Synchronisation

3.3.1 The Yaws Webserver

Yaws [73, 52] is a high-performance, actor-based HTTP webserver written in Erlang. For every client connection, this server assigns a dedicated handler actor that listens for and services HTTP client requests. At its core, its implementation fundamentally relies on the lightweight nature of Erlang processes to be able to efficiently handle a vast amount of client connections.

The Yaws protocol for establishing client connections is depicted in Fig. 3.5. This protocol relies on an acceptor component which upon creation, it spawns a connection handler to be assigned to the next client connection. Subsequently, the acceptor blocks waiting for messages in its mailbox, while the unassigned handler waits for the next TCP connection request. Clients send connection requests through standard TCP ports (1), which are received as messages in the handler’s mailbox. The current handler accepts these requests by reading the respective message from its mailbox and (2) sending a message containing its own pid and the port of the connected client to the acceptor. This acts as a notification that the handler is now engaged in handling the connection of a specific client. Upon receiving the message, the acceptor unblocks, records the information sent by the handler and (3) spawns a new handler listening for future connection requests from other clients.

Once it is assigned a handler, the connected client then engages directly with it by using (4) standard HTTP requests; these normally consist of six (or more) HTTP headers containing information such as the client’s User Agent, Accept-Encoding and the Keep-Alive flag status. It is worth highlighting that the HTTP request information is not sent in one go, but follows a protocol of messages: it starts by sending the http.req, followed

![Figure 3.5: Yaws client connection protocol](image-url)
by six (or more) \texttt{http-header} messages containing client information, terminated by a final \texttt{http-eoh} message. The dedicated connection handler inspects the request and the client information received in the headers, and services the respective HTTP request accordingly.

Earlier versions of the Yaws webserver were found to be suffering from a Directory Traversal Vulnerability \cite{Yaws} which was also reported on the reputable exploit-db website \cite{ExploitDB}. This vulnerability permits for attackers to perform a \textit{Dot-dot slash} attack in which a malicious client provides the webserver with an invalid, malformed URL, containing a sequence of dots and slashes, in an attempt to exploit the vulnerability. This allows external clients to navigate to sensitive directories, pertaining to the server, such as the \texttt{System32} directory.

### 3.3.2 Monitoring for safety properties in Yaws

Despite the simplicity and elegance of detectEr's sHML specification language, it turns out that it provides enough expressivity to specify numerous safety properties for the Yaws webserver. For brevity in this section we only present two of these properties, and in appendix Sec. \ref{appendix:more_properties} we present another two properties.

**Detecting Malicious Requests**

In the first property we assume the existence of the \texttt{isMalicious(..)} predicate to define \textit{Property 4} (below). This (decidable) predicate can determine whether the client will engage in a security-breaching activity by analysing the 6 HTTP headers sent to the handler.

\textbf{Property 4.} Every time a client connection is established (determined from message (2) of Fig. \ref{fig:connection_establishment}), and the assigned handler receives an HTTP-Get request followed by 6 HTTP headers (h1 to h6) and terminated by the end-of-headers notification, then the communicated headers must not amount to a potentially security-breaching request as determined by the predicate \texttt{isMalicious([..]).}
3. Assessing the Impact of Increasing Monitor Synchronisation

Using `detectEr`'s formal logic presented in Fig. 2.10 (see Sec. 2.4.1), we can now formalise Property 4 as formula (3.3).

\[
\begin{align*}
\max X ( & \\
[& \text{acceptor?}(\text{hPid}, \text{next}, _)] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, http_req, GET, _ -}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h1}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h2}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h3}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h4}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h5}\}))] \\
[& \text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, h6}\}))] \\
\text{if} (\text{isMalicious}(h1, h2, h3, h4, h5, h6) \text{then } \text{ff} \\
\text{else } [\text{hPid} \rightarrow \text{ret}((\text{yaws}, \text{do_recv}, 3, \{\text{ok, http_eoh}\})]) X)
\end{align*}
\] (3.3)

The logical formula stated in (3.3) specifies this property as a recursive formula which pattern matches the assigned handler to term variable `hPid`. This value is then used to pattern match with the header term variables, `h1` to `h6`, for every iteration of the HTTP request protocol. An if-statement is then used to evaluate the `isMalicious(..)` predicate to determine whether the received HTTP headers pose a security threat or not.

We note that, whereas the handler messages to the acceptor are observed directly (i.e., by using input action `[acceptor?{hPid, next, _}]`), the client HTTP messages received by the handler have to be observed indirectly through the return values (of the form `{ok, header}`) of the invoked function `do_recv`, which is defined in module `yaws` with arity 3. Instrumentation allowing a direct observation of these actions is complicated by the fact that the client TCP messages are sent through functions from the `inet` Erlang library, which is part of the Erlang Virtual Machine kernel [58, 7].

### Detecting Directory Traversal Exploitation

In a similar manner we specify yet another property for detecting behaviour that may exploit the Directory Traversal Vulnerability, found in earlier versions of the software, as described in Sec. 3.3.1. For the intents and purposes of this case study, we aim to detect such possible attacks by monitoring for external client requests and compare the requested urls to a white-list. We assume that external clients are only allowed to request for two files, namely “pig.png” and “site.html” as specified in safety property 5.
3. Assessing the Impact of Increasing Monitor Synchronisation

**Property 5.** Every time a client connects, and the assigned handler receives an HTTP-Get request for a specific file stored on our server, followed by 6 HTTP headers (h1 to h6) and the end-of-headers notification, then the requested file can only refer to either for “pig.png” or “site.html”.

We formalise this property (i.e., Property5) as recursive formula (3.4) which binds the assigned handler with term variable \( hPId \), and then uses this value to pattern match with the HTTP-GET request, the 6 headers, and the ending header \( \text{http}_\text{eoh} \). The file requested in the HTTP-GET request is pattern matched and stored in the \( \text{path} \) term variable. The matched path value is then inspected by an if-statement that checks whether the requested path points either to “pic.png” or “site.html”. If the path is found to be pointing to any of these two files, the property immediately checks that the server actually sends the requested file to the client and recurs on formula variable \( X \); otherwise, a property violation is flagged.

\[
\begin{align*}
\text{max} X ( & \\
& [\text{acceptor}! [hPId, \text{next}, \_]] \\
& \quad [hPId \triangleright \text{ret}([\text{yaws.do_recv}, 3, \{\text{ok, \{\text{http}_\text{req, 'GET', \{\text{abs}_\text{path}, \text{path}\}, \_}\}}]])] \\
& \quad [hPId \triangleright \text{ret}([\text{yaws.do_recv}, 3, \{\text{ok, \_}\}])] \\
& \textrm{max} X \quad \vdots \\
& \quad [hPId \triangleright \text{ret}([\text{yaws.do_recv}, 3, \{\text{ok, \text{http}_\text{eoh}}\}])] \\
\text{if} \quad \text{path} == \text{"/pic.png" \ or else \ path} == \text{"/site.html" \ then} \\
& \quad [hPId \triangleright \text{call}([\text{yaws.sendfile, send, [\_ \_}, \_ \_\_]]) \ X] \\
& \quad \text{else ff} \quad ) \\
\end{align*}
\]

(3.4)

Adapting these Properties for Hybrid Monitoring

To be able to monitor for these properties using our hybrid approach, we use the extended syntax (introduced in Sec. 3.2.1) to reformulate security properties (3.3) and (3.4) whereby we require violation detections to be synchronous i.e., using \( \text{sff} \) instead of \( \text{ff} \). For instance, from formula (3.4) we obtain the hybrid formula (3.5) stated below.
max $X$ ( 
[acceptor!{hId, next, .}] 
[hId ▽ ret({yaws, do_recv, 3, [ok, {http_req, 'GET', {abs_path, path}, _}]}),]  
[hId ▽ ret({yaws, do_recv, 3, [ok, h1]})]  

;  
[hId ▽ ret({yaws, do_recv, 3, [ok, h6]}),]  
[hId ▽ ret({yaws, do_recv, 3, [ok, http_eoh]}),]  

if $path == "/pic.png"$ or else $path == "/site.html$ then  
[hId ▽ call({yaws_sendfile, send, [_, path, _, _]}),]  
else $sff$
)

All of the other properties used in our experiments where reformulated in the same manner as per hybrid formula (3.5).

### 3.3.3 The Performance Results

We measure the respective overheads resulting from the three monitor instrumentation techniques that are now supported by detectEr, over Yaws for varying client loads, in terms of:

- the average number of CPU cycles required per Yaws client request;
- the average memory utilisation for the Yaws webserver to respond to batches of client requests; and
- the average response time of the (monitored) Yaws server, i.e., the time taken for the server to respond to a client request.

**Remark 3.** In our experiments we utilise a version of detectEr, presented in [15], that provides two monitoring optimisations — a static optimisation and a dynamic optimisation, along with the original unoptimised monitor synthesis technique given in [39]. These techniques only optimise the way that the concurrent monitors are synthesised and the way they operate at runtime, but they are independent from the system instrumentation used for event reporting. This means that they can be applied to every monitoring instrumentation technique explored so far (i.e., asynchronous, synchronous and hybrid...
3. Assessing the Impact of Increasing Monitor Synchronisation

instrumentation). This enables us to ensure that the performance discrepancies obtained in our experiments persist regardless of how the synthesised concurrent monitors are optimised. This allows us to rule out that the discrepancies are caused due to unoptimised monitors. We also provide a brief overview of these optimisations in Appendix Chapter A.

The experiments were carried out on an Intel Core 2 Duo T6600 processor with 4GB of RAM, running Microsoft Windows 7 and EVM version R16B03. To further confirm the discrepancies between synchronous, hybrid and asynchronous monitoring we run our test-suite and take separate batches of results using (i) unoptimised monitors, (ii) statically optimised monitors and (iii) dynamically (fully) optimised monitors.

For each property (in our test-suite) and each client load, we take five sets of readings and then average them out. Since results pertaining to a specific monitor optimisation do not show substantial variations when using different properties, we again average the results across all properties and compile them as graphs for each optimisation. We thus obtain the graphs shown in Figures 3.6 to 3.8 — the same results are also compiled as tables in appendix Sec. B.2.

From the graphs in Figures 3.6 to 3.8 we note that regardless of how the monitors are optimised, memory and CPU utilisation suffer from a marginal difference in which the hybrid monitoring overheads generally lie midway between synchronous monitoring – which scored the worst; and asynchronous monitoring – which gave the best performance results. A clearer, and more substantial difference is seen in the average time that the server takes to respond to each client request when it is under a certain load. Fig. 3.6 shows that unoptimised monitors suffer from a substantial response difference of about 46ms when a synchronously monitored instance of Yaws is placed under the load of 2000 client requests, as compared to its asynchronous counterpart. By contrast hybrid monitoring provided timely detections by suffering only from a 9ms response overhead increase (for the same load) when compared to asynchronous monitoring.

2Providing low average response times is imperative in reactive systems (such as web servers) to make sure that the client is served as soon as possible.
3. Assessing the Impact of Increasing Monitor Synchronisation

![Graphs showing performance impact analysis](image)

Figure 3.6: Performance impact analysis (using unoptimised monitors)
3. Assessing the Impact of Increasing Monitor Synchronisation

Figure 3.7: Performance impact analysis (using statically optimised monitors)
3. Assessing the Impact of Increasing Monitor Synchronisation

Figure 3.8: Performance impact analysis (using fully optimised monitors)
Furthermore, Figures 3.7 and 3.8 demonstrate that even though the monitoring optimisations [15] help in reducing the overall overheads, synchronous instrumentation monitoring still fared the worst when compared to the rest. In fact this suffers from a 19ms difference (compared to asynchronous) when using statically optimised monitors to synchronously monitor a Yaws instance placed under the load of 2000 client requests, and a 9ms difference with dynamically optimised monitors. Contrasting this, the hybrid approach only suffers from a 4ms and 1.5ms response time difference for statically optimised and dynamically optimised monitors respectively.

We therefore conclude that regardless of any optimisation used by the synthesised monitors, synchronous instrumentation monitoring incurs substantial overheads when compared to its asynchronous counterpart. By contrast, the hybrid instrumentation results show that timely detections can still be achieved with overheads that are higher yet closer to that of asynchronous monitoring.

The reason for this performance degradation is apparent from Fig. 3.2 and Fig. 3.4, where one can notice that for every synchronous monitored event, the system execution incurs a responsiveness penalty waiting for the monitor to acknowledge back. A further penalty is also incurred since synchronous event reporting requires executing additional code (e.g., to generate nonces) when compared to its asynchronous counterpart. Hence hybrid monitoring performs better than synchronous instrumentation monitoring as the former performs less synchronisations. For the same reason, hybrid monitoring still imposes a slightly higher degree of overheads than asynchronous monitoring as it still needs to apply a small level of synchrony in order to achieve timely detections.

3.4 Conclusion

In this chapter we have analysed the design space for introducing synchrony on top of an inherently asynchronous platform. The primary objective of this investigation was to identify a synchronisation technique that allows for a tighter monitoring control to be achieved efficiently. This contributes towards developing an effective, yet efficient RA
3. Assessing the Impact of Increasing Monitor Synchronisation

We started by devising a handshaking protocol for achieving synchronous event monitoring on actor-based Erlang systems which required refining an AOP framework for Erlang systems to instrument the protocol in existing systems. Using this handshaking protocol we introduced a coarse-grained synchronous instrumentation monitoring approach which synchronised for every reported event. Following this we reduced the level of synchronisation by limiting it to critical actions i.e., system actions that can directly contribute to a violation. This lead to creating a hybrid monitoring approach capable of switching between synchronous and asynchronous event reporting, while still ensuring timely detections when required. This however required us to extend the detectEr’s specification language.

After integrating the synchronous and hybrid monitoring approaches in detectEr we carried out a systematic assessment wrt. Yaws, regarding the relative overheads incurred by different instrumentation techniques within an actor setting. The results of this impact assessment show that the hybrid technique yielded response times that are lower than those of synchronous instrumentation approach.

Therefore, the conducted impact analysis enables us to conclude that since synchrony is not natively supported by asynchronous, actor-based systems, this induces inevitable synchronisation overheads. However, our analysis also allows us to conclude that although the incremental synchronisation methodology used in our hybrid technique still introduces a certain degree of overheads, these can be minimised and kept to a feasible standard if synchronisation is utilised on a by-need basis; i.e., only when it is strictly required (e.g., to timely achieve detections or to effectively apply adaptations).

In the next chapter we therefore build upon the incremental synchronisation methodology that we have identified by this impact assessment, in order to implement an efficient yet effective Runtime Adaptation framework for actor systems.

---
3A stable release of this version called detectEr 2.0 is also accessible from https://bitbucket.org/casian/detecter2.0
4. Language Design, Implementation and Formalisation

In this chapter we present the design and implementation details about how we develop our runtime adaptation framework on top of the detectEr RV tool. Based on the implementation constraints we then develop a formal operational model which provides a higher level description of how our RA scripts behave at runtime. We divide this chapter into two parts: in the first part (Sec. 4.1 to 4.4) we analyse the implementation challenges, while in the in the second part (Sec. 4.5) we explain how we developed the formal model for our RA scripts.

More specifically, in Sec. 4.1 we identify a number of adaptation actions that should be applicable on any arbitrary actor system, regardless of the design and coding practices used in its implementation. We also articulate the constraints that the actor model imposes which makes it more challenging to implement such adaptation actions. We therefore provide a preliminary investigation of how we can overcome these challenges.

In Sec. 4.2 we present our design decisions for integrating the identified adaptations within the logic used by detectEr, thereby enabling users to specify runtime adaptation monitors for mitigating property violations. This is followed by Sec. 4.3 in which we expand on the preliminary investigation given in Sec. 4.1 and provide an overview of how we implement the synchronisation protocol required by our runtime adaptation framework. We conclude the first part by showing how one of the RV scripts written for Yaws
(used in the evaluation of Chapter 3), can easily be converted into an RA script for our new framework.

Finally in Sec. 4.5 we focus on developing a formal model that allows for predicting the runtime behaviour of an RA script without actually running it. This allows us to analyse the behaviour of our scripts without having to deal with the complexities of the implementation, thereby enabling us to further understand the possible errors that erroneous RA scripts may introduce.

4.1 Identifying Adaptation Actions within the Constraints of Actor Systems

Actor design practices such as the fail-fast design pattern and systems structured using supervision-trees, permit for the creation of self-adaptive actor systems whereby supervisor actors are purposely developed to intervene and mitigate cases when an actor crashes abnormally. As misbehaviour does not necessarily cause an actor to crash, these design strategies are limited in the type of misbehaviour that they can detect (i.e., detections only happen when an actor crashes). One way how to to overcome this limitation is to employ an RV monitor which is able to deliberately interfere by crashing certain system actors whenever it detects erroneous behaviour. This would in turn trigger the system’s built-in self-adaptive functionality which mitigates the crashed actors.

Example 4.1.1. Assume a simple self-adaptive actor system illustrated in Fig. 4.1 (below). This system consists in four components, namely A, B and C which are the actual worker components, along with component S which is their supervisor. The sole purpose of this supervisor is to apply the necessary mitigation when either one (or more) of the worker components crash.

As explained by Fig. 4.1 this simple RA monitor (1) receives and analyses the system events, and upon detecting a violation it (2) deliberately terminates one of the system’s components (i.e., component C) instead of flagging the violation. Upon crashing,
Figure 4.1: Triggering the self-adaptive functionality of a system through Runtime Monitors.

(3) this component sends a link message to its supervisor, $S$, which in turn (4) applies the necessary mitigation.

Although this straightforward RA mechanism is already quite useful and effective, it relies on the assumption that the monitored actor system already contains a certain degree of self-adaptive capabilities. In cases where a monitored actor system lacks (or has limited) self-adaptive functionality, killing an actor would not trigger any type of mitigation mechanism. At best this could be used to permanently switch-off redundant or useless parts of the system such as orphaned actors \cite{69} i.e., actors that are no longer in use but for some reason did not terminate. At worst, however, killing actors in a non-adaptive system might cause it to become unresponsive.

We must therefore look into other adaptation techniques that could enable us to mitigate erroneous actors though high-level actor-manipulation techniques, that can be applied regardless of the design and implementation of the system under scrutiny. In other words our adaptations should be applicable to any system actor irrespective of the code that it executes, and of the way that it is linked with other actors. However, implementing such complex adaptations which are agnostic of the actor’s code, is quite challenging due to certain restrictions imposed by the actor model and the host language (in our case Erlang).
4. Language Design, Implementation and Formalisation

4.1.1 Identifying ideal Runtime Adaptation Actions

An RA framework for actor systems should provide generic adaptations that allow the specifier to manipulate a number of architectural aspects pertaining to actor systems. To identify what these aspects really are, we recall the basics of the actor model (presented in Sec. 2.3), which we summarise in the following three points:

(i) An actor is a concurrent process.
(ii) An actor has a mailbox for asynchronous interaction (through message passing).
(iii) Actors can be structured (i.e., related through linking).

As stated in (i), since actors are essentially processes and given that we want to keep our mitigation techniques as generic as possible, the only two types of generic process manipulations that can be applied upon a black-box process (i.e., without knowing what it executes) are (a) Process Terminations and (b) Process Restarts.

(a) **Process Terminations:** These operations should be carefully used to terminate non-crucial misbehaving actors, such that the actor system is able to carry on free of the erroneous actors, even though some of its functionality might be unavailable.

**Example 4.1.2.** Recall the system illustrated in Fig. [1.1] for which we have already formalised RV Property [4] in terms of detectEr’s logic, as formula (2.1) in Ex. 2.4.1 (restated below).

\[
\varphi \overset{\text{def}}{=} \max Y. [i?\{inc, x, y\}] \left( ([j\triangleright y!\{res, x + 1\}] Y) \land ([\_\triangleright y!\text{err}] \text{ff}) \right) \tag{2.1}
\]

Based on (a) we reformulate (2.1) as (4.1) whereby the script now includes a the kill adaptation which terminates the actor that sends the error message err to the connected client. Note that we now map this erroneous actor to term variable \(z\) so that we are able to apply the kill adaptation upon it.

\[
\max Y. [i?\{inc, x, y\}] \left( ([j\triangleright y!\{res, x + 1\}] Y) \land ([\_\triangleright y!\text{err}] \text{kill}(z) Y) \right) \tag{4.1}
\]

As in our example system (given in Fig. [1.1] we do not assume any sort of built-in self-adaptive functionality (i.e., no linking, trapping, supervisors, etc.), in formula (4.1) we
eliminate future occurrences of this error from being made by the same erroneous actor, by permanently switching off this actor. Although this termination restricts the overall functionality of our system, we still retain the rest of the system’s functionality as the other concurrent actors are not effected by the applied adaptation.

(b) Process Restarts: These actions should be used to restart more crucial components in an attempt to remove/reduce any side-effects (that may cause further errors) incurred due to the detected misbehaviour, by reinitialising them afresh.

Example 4.1.3. For the example system in Fig. 1.1 we can now extend formula (2.1) (restated in Ex. 4.1.2) with the restart adaptation in (4.2).

\[
\max Y. ([i?\{inc, x, y\}] ([j>y!\{res, x + 1\}] Y) \& ([z>y!err]\text{restr}(i, z)\text{tt}) )
\]  

We use the restart adaptation to reinitialise both the common-interface actor \(i\), and also the actor mapped to term variable \(z\), in order to mitigate the invalid behaviour which lead to producing error \(err\). In this way we still retain the full functionality of the system while we reduced the possible side-effects inflicted on the mitigated actors, by restarting them from their point of initialization.

Furthermore from architectural aspect (\(ii\)), we know that actors are not just concurrent process, but they are also interactive processes. In fact actors are able to interact asynchronously by depositing messages in the private mailbox of other actors. Due to this method of interaction we identify another generic adaptation, namely:

(c) Message Interception: These adaptations should be used to intercept and manipulate messages that might have been erroneously exchanged due to the detected violation, thereby preventing them from causing further problems.

Example 4.1.4. Assuming that we can use adaptation \(prg\) to purge all the messages in an actor’s mailbox, we now develop (4.3) as an extension of (2.1).

\[
\max Y. ([i?\{inc, x, y\}] ( [j>y!\{res, x + 1\}] Y) \& ([z>y!err]\text{prg}(z)Y) )
\]
In (4.3) we use the prg adaptation to flush the mailbox contents of the actor mapped to \( z \), with the aim of removing other conflicting messages that the erroneous actor (mapped to \( z \)) might have received prior to sending the error message to the client.

Furthermore, as by (iii) we know that these communicating actors can also be structured and related, we identify yet another generic adaptation that allows us to manipulate this aspect of actor systems:

\[ (c) \textbf{Process Restructuring}: \] There are various ways how actor systems can be structured including: how the actors are linked together e.g., actor \( A \) is linked to actor \( B \) but not to \( C \); the type of linking between actors e.g., unidirectional or bidirectional, trapped or untrapped; and the way actors are identified e.g., some actors are registered with a unique name while others are not. Restructuring should therefore allow us to manipulate these structural aspects.

\textbf{Example 4.1.5.} Recall from Sec. 2.3 that when a process dies, every process linked to it dies as well (unless trapping is used). Assuming that the common-interface (from Fig. 1.1) is linked to the decrementor and incrementor components, in (4.4) we use the unlink restructuring adaptation to remove the link between the interface, \( i \), and the actor mapped to \( z \).

\[
\max Y. \left[ i ? \{ \text{inc}, x, y \} \right] \left( \left[ j \triangleright y! \{ \text{res}, x + 1 \} \right] Y \right) \& \left( \left[ z \triangleright y! \text{err} \right] \text{unlink}(i, z) Y \right) \quad (4.4)
\]

In this way, if the actor mapped to \( z \) crashes as a result of producing error \text{err}, at least the other linked components can still operate. Hence rather than losing all the system’s functionality, we allow for the system to remain partially operational.

The identified adaptations (a), (b), (c) and (d) are essentially \textit{categories} of adaptations rather than actual adaptation actions. We also conjecture that \textit{instances} of these categories can be implemented in various ways, but ultimately provide the same overall effect e.g., one instance of the restructuring category is used to \textit{add links} between actors, while another instance is used to \textit{remove links} instead; these reside in the same category as they ultimately both manipulate the structure of an actor system, even if they do it in a different way.
Remark 4. The overall effects of different categories can also be combined to obtain even complex adaptations. A case in point is the silent-kill adaptation which despite being an instance of category (a) (i.e., process terminations), it is also able to manipulate the structural aspects of an actor system. In fact whenever the silent-kill is applied to an actor \( i \), it removes any link related to \( i \) prior to terminating it. It is therefore able to suppress the system’s built-in self-adaptive functionality (if any) by removing the links that alert the other actors about the terminated actor, and thus prevent the system from attempting to apply its own mitigation.

Example 4.1.6. With the identified adaptation categories we are now in a position to formally specify RA \( \text{property}2 \) (see Ex. 1.1.1) as an extension of RV \( \text{property}1 \) which we have already formalised as formula (2.1) in Ex. 2.4.1. We thus obtain formula (4.5):

\[
\max Y. [i?\{\text{inc}, x, y\}] \left( ([j \triangleright y!\{\text{res}, x + 1\}] Y) \& ([z \triangleright y!\text{err}] \text{restr}(i) \text{prg}(z) Y) \right)
\]

(4.5)

The specifier presumes that the error (which may arise after a number of correct interactions) is caused by the interface actor \( i \) (as shown in Fig. 1.1, where an \text{inc} request is erroneously forwarded to the decrementor actor \( k \)) — the specifier may, for instance, have prior knowledge that actor \( i \) is a newly-installed, untested component. The monitor thus restarts actor \( i \) by using adaptation \text{restr}(i), and empties the mailbox of the backend server through adaptation \text{prg}(z), as this may contain more erroneously forwarded messages. Note that the actor to be purged is determined at runtime, where term variable \( z \) is bound to identifier \( k \) from the previous action \([z \triangleright y!\text{err}]\). Importantly, note that in the above execution (where \( k \) is the actor sending the error message), actor \( j \)'s execution is not affected by any adaptation action taken.

4.1.2 The Implementation Constraints of Adaptation Actions

Although some adaptations are supported by the host language, other adaptations need to be implemented. Implementing adaptations pertaining to some of the identified categories is not always straightforward, as the actor-model and the host language restrict the way that actors can externally effect other actors. In the case of Erlang, as illustrated
in Fig. 4.2, an actor $A$ is not allowed to externally interrupt the operation of another actor $B$, thereby forcing it to execute interrupt service routine (ISR) which invokes the required adaptation function. Although interrupts and ISRs are commonly used in concurrent and multithreaded systems [69], they are strictly abolished by the actor-model and Erlang.

![Diagram of applying adaptations using interrupts](image)

**Figure 4.2:** Applying adaptations using interrupts.

Instead, as shown in Fig. 4.3, in the actor-model the adapter actor $A$ must encode and adaptation request as a message and deposit it in the mailbox of adaptee $B$, requesting it to execute the required adaptation function. The recipient actor, $B$, must then read the message from its private mailbox, interpret it and willingly execute the required function (it might choose not to).

![Diagram of applying adaptations encoded as messages](image)

**Figure 4.3:** Applying adaptations encoded as messages.

The only type of external actions that are natively supported by the Erlang OTP libraries [58] are actor killing and linking. This allows us to augment detectEr’s RV
monitors with flavours of adaptation categories (a) and (d) by enabling them to execute
the respective OTP functions over the necessary system actors.

Remark 5. Although these natively supported adaptations can also be used with asyn-
chronous monitoring, recall that this suffers from late detections. This means that even
though we are able to apply these adaptations, it does not necessarily mean that they
can be effectively applied to prevent or properly mitigate any possible side-effects. For
instance, recall that in formula (4.1) we use the kill operation to terminate the actor that
sends the err message to the client. However, if this actor is not terminated in a timely
manner it would still be able to progress and perform further operations, that may lead to
more serious consequences, prior to being killed. Similarly, in the case of formula (4.4),
if the interface actor i and the erroneous actor mapped to z are not unlinked on-time, ac-
tor z might progress and crash (thereby causing the linked actors to crash as well) before
the monitor manages to remove the links.

The Erlang OTP libraries, however, still do not provide native mechanisms for restart-
ing actors (i.e., implementing (b)) or for reading and modifying the mailbox contents of
another actor (i.e., to implement (c); in fact this is strictly abolished). Hence as
outlined in Fig. 4.3, to implement these adaptations we have to somehow ‘trick’ the
adaptee into providing the monitor with enough control to willingly perform these more
complex operations.

It turns our that synchronisation not only enables us to address the issue of apply-
ing adaptations effectively, outlined in Remark 5 (as explored in Chapter 3), but also
provides the monitor with a tighter control over the system actors. This allows for the
implementation of more complex adaptations such as those pertaining to categories (b)
and (c). The general idea for implementing these adaptations is to use synchronisation
as a form of local anesthetic which places the necessary system actors in a state of con-
trolled dormancy that allows the monitor to operate upon them.

More specifically, as illustrated in Fig. 4.4 whenever a system actor is blocked by
the instrumented synchronisation protocol, this does not only wait for the monitor’s ac-
knowledgement message, but also waits for adaptation requests encoded as messages. Upon receiving an adaptation request, e.g., `restr`, the code instrumented in the system actor can then analyse the request and carry out the necessary procedure for applying the requested pre-instrumented adaptations. As the adaptations are executed within the system actor’s own thread of execution, these adaptation functions would also have full access to the actor’s private mailbox and state (i.e., its process dictionary), thereby allowing for more intrusive operations, such as message interception, to be performed. The required adaptations can therefore be implemented as regular functions which are then instrumented in the system such that they can be invoked by the instrumented synchronisation mechanism.

However, if an adaptation message is sent to an active system component (i.e., an actor which was not blocked by our instrumentation), the actor might suffer the following consequences:

- it might not even read the request message, hence the adaptation is *not conveyed*;
- it might read the message while it is performing other operations and *ignore* it, as it would not know what it means;
- the message might be read during the actor’s normal execution and cause *conflicts* leading to a *crash*.

It is therefore imperative to make sure that the respective actors are suspended by our
instrumentation prior to applying these adaptations.

Hence, due to these implementation constraints, we classify the implementable adaptations into the following two classes:

- **Asynchronous Adaptations**: these can be administered on the respective actors while they are executing, typically these include natively supported actions.

- **Synchronous Adaptations**: the actors requiring these adaptations must be suspended by our instrumentation prior to applying them.

Based on the adaptation categories identified in Sec. 4.1.1 and the adaptation classes presented above we create Table 4.1. This presents a matrix of adaptations denoting the categories that can be applied synchronously, asynchronously or both. In general, asynchronous adaptations e.g., actor linking and killing, can also be applied synchronously.

| Classes       | (a) Terminations | (b) Restarts | (c) Interceptions | (d) Restructuring |
|---------------|------------------|--------------|-------------------|------------------|
| Synchronous   | ✓                | ✓            | ✓                 | ✓                |
| Asynchronous  | ✓                | ✓            | ✓                 | ✓                |

Table 4.1: A classification for Synchronous and Asynchronous Adaptation Categories.

### 4.2 Designing the Runtime Adaptation Mechanism

As synchronous adaptations require suspending the adaptee prior to applying the adaptation upon it, we employ an incremental synchronisation mechanism for *gradually suspending only* the actor executions of interest as the property is being monitored for, *i.e.*, without suspending any of the other actors. We opt for incremental synchronisation based on the results we obtained from our impact assessment carried out in Chapter 3. This demonstrated that incremental synchronisation is able to keep synchronisation overheads under control by applying synchrony on a *by-need* basis, thereby also allowing for feasible runtime adaptation. Hence, inspired from synchronous necessities (*i.e.*, \([e]\)) in our hybrid monitoring scripts (see Sec. 3.2.1) we extend necessity formulas with a *synchronisation modality*, \([e]^{\rho} \varphi\), where \(\rho\) ranges over either \(b\) (blocking): stating that the
subject of the action (i.e., the actor that committed the action) is suspended if its pattern is matched; or a (asynchronous): stating that the action subject is allowed to continue executing asynchronously, even when the pattern is matched.

We also recall that if the necessity formula \( [e] \varphi \) mismatches with a trace action, its observation terminates due to the trivial satisfaction of the formula. In our case this also means that the synchronous adaptations contained the continuation \( \varphi \) are never administered. Hence, any actor executions that were previously blocked so as to make these synchronous adaptations applicable are no longer required to remain blocked. We thus provide a mechanism for releasing the actors blocked so far by further extending necessity formulas with a list of actor references, RL. This list denotes the (blocked) actors to be released in case the necessity pattern \( e \) mismatches, \( [e]_{RL}^\varphi \).

**Example 4.2.1.** Recall formula (4.5) from Ex. 4.1.6 (restated below).

\[
\max Y. [i?\{inc, x, y\}] \left( ([j \triangleright y ! \{res, x + 1\}] Y) \& ([z \triangleright y ! err] restr(i) prg(z) Y) \right) \tag{4.5}
\]

We augment the necessity operations in (4.5) with the required synchronisation modalities and release lists so to obtain (4.6).

\[
\max Y. [i?\{inc, x, y\}]_e^b \left( ([j \triangleright y ! \{res, x + 1\}]_e^a Y) \& ([z \triangleright y ! err]_b restr(i) prg(z) Y) \right) \tag{4.6}
\]

Formula (4.6) states that since actor \( i \) might require being adapted (by \( restr(i) \)), then this must be blocked by setting the modality of necessity declared prior to the adaptation to blocking. In this case we block actor \( i \) whenever it receives an increment request \([i?\{inc, 4, h\}]\), i.e., by using blocking necessity \([i?\{inc, x, y\}]_e^b \). In case a result, e.g., \([res, 5]\), is sent to the client mapped to \( y \) instead of an error message \( err \), this means that this event would match the pattern of necessity \([j \triangleright y ! \{res, x+1\}]_e^a \).

However, this also implies that necessity \([z \triangleright y ! err]_b \) would become trivially satisfied and hence would not proceed to adapting actor \( i \) with \( restr(i) \), as this is no longer necessary. Hence, our formula must release actor \( i \) upon its trivial satisfaction thereby allowing it to receive further increment requests. We therefore add \( i \) to the release list of

\^1Note that this might cause the entire system to deadlock; we however explain this later on in Remark [6] and Ex. 4.2.2.
this necessity, i.e., $[z \triangleright y \text{err}]^b_i$. Moreover we set the synchronisation modality of this necessity to blocking, i.e., $[z \triangleright y \text{err}]^b_i$ in order to suspend the actor mapped to variable $z$ prior to purging its mailbox.

Since adaptations in a script may be followed by further observations, we also require a similar release mechanism for adaptation actions, i.e., adaptation(AL)$_{RL}$, where the actors defined in the release list, $RL$, are unblocked after the adaptation is administered to the actors defined in the adaptation list, $AL$.

**Remark 6.** Although minimally intrusive, the expressivity of our mechanism for incremental synchronisation relies on what system actions can be observed (i.e., the level of abstraction at which the system is monitored). For instance, recall the server system depicted in Fig. 1.1. If monitored from an external viewpoint, the communications sent from the interface actor, $i$, to the backend incrementor and decrementor actors, $j$ and $k$, are not visible (according to [39], they are seen as internal $\tau$-actions). However, for observations required by properties such as (4.5), we would need to block actor $i$ only after it sends a message to either of the backend actors—otherwise the entire system blocks (such as in (4.6)). This requires observing the system at a lower level of abstraction, where certain $\tau$-actions are converted into visible ones e.g., the instrumentation used by detectEr allows us to observe internal actions such as function calls or internal messages sent between actors as discussed for Fig. 1.1.

**Example 4.2.2.** In (4.7) we extend (4.6) with the observation of an internal communication action (i.e., $i \triangleright \{\text{inc, x, y}\}$) for blocking purposes (as explained in Remark 5), and also with release lists for adaptations restr and prg.

\[
\varphi' \overset{\text{def}}{=} \max Y. \left[ i ? \{\text{inc, x, y}\}^a \cdot \begin{array}{c}
\left[ i \triangleright \{\text{res, x + 1}\}^a \cdot Y \right] \\
\left[ z \triangleright y \text{err}\right]^b \cdot \text{restr}(i)_x \cdot \text{prg}(z)_x \cdot Y
\end{array} \right]
\]  

(4.7)

After asynchronously observing $i ? \{\text{inc, v, h}\}$ (for some $v$, $h$), the respective monitor synchronously listens for an internal communication action (i.e., $[i \triangleright \{\text{inc, x, y}\}]^b_x$) from $i$ to some actor with the same data, $\{\text{inc, v, h}\}$; if this action is observed, the subject of the action (i.e., $i$) is blocked. If the subsequently observed action is an error reply, $z \triangleright h \text{err}$
originating from an actor bound at runtime to \( z \), we block this and start applying the necessary synchronous adaptation actions, i.e., \( \text{restr}(i) \) and \( \text{prg}(z) \). Note that the last adaptation action releases the two blocked actors \( i \) and \( z \) before recurring; similarly the necessity formula for the error reply releases the blocked actor \( i \) if the respective action, \( [z \triangleright y! \text{err}]^b_i \), is not matched.

4.3 Implementing the Runtime Adaptation Protocol

We implement our runtime adaptation protocol (illustrated in Fig. 4.5 below) as an extension of the incremental synchronisation technique introduced in our hybrid monitoring protocol (see Sec. 3.2). The system instrumentation for the new protocol remains more or less the same as in the hybrid protocol. In fact asynchronous actions (i.e., \( [e]^a_{\text{RL}} \)) inject advice functions that send monitoring messages, containing the event details and a null nonce, without blocking the action’s subject. By contrast, blocking actions (i.e., \( [e]^b_{\text{RL}} \)) generate and send a fresh nonce along with the event details to the monitor, and subsequently cause the instrumented actor to block waiting for a releasing acknowledgement message from the monitor. While waiting, an instrumented blocked actor can now also receive synchronous adaptation requests.

Upon receiving an adaptation request message containing: a valid nonce, and an adaptation tag denoting a valid pre-instrumented adaptation function (e.g., \( \text{restr} \) to denote the restart function), the system side instrumentation sets the requested adaptation as next-in-line for execution. The requested adaptations are only applied if and when the blocked system actor receives the releasing acknowledgement.

To allow for the monitor to adapt and release system actors, even after receiving further event messages, the monitoring loop now maintains a map of nonces. Whenever the monitor receives an event message containing a non-null nonce and the subject’s process id, it adds the received nonce as a new entry to its map by using the subject’s id as key; it then proceeds by handling the event. As event handling now also includes applying adaptations (i.e., \( \text{adaptation(AL)}_{\text{RL}} \)), the monitor may proceed by issuing the
4. Language Design, Implementation and Formalisation

necessary adaptation actions denoted in the adaptation list AL (if any), and then releases the actors enlisted in the release-list RL. To issue synchronous adaptations and release acknowledgements, the monitor uses the process ids enlisted in AL and RL as keys for retrieving the associated nonces from its map. After applying the necessary adaptions and releases, the monitor removes the map entries associated with the released actors. It finally reiterates and waits for the next system event.

4.3.1 Implementing Pre-instrumented Synchronous Adaptations

Although Erlang already natively implements some of the adaptations that we categorised in Table 4.1 as part of our prototype we implement more complex and effective synchronous adaptation actions. More specifically, we tried to implement a representative adaptation action for each synchronous adaptation category presented in Table 4.1. These implementations can be applied over instrumented system actors using the RA protocol implementation outlined in Fig. 4.5. We therefore implement the following

Figure 4.5: A high-level depiction of the Runtime Adaptation protocol.

---

System

... Actor i commits event e1;  
i gets data d1 from e1;  
i blocks on nonce(e1);  
... Actor j commits event e2;  
j gets data d2 from e2;  
j blocks on nonce(e3);  
... Actor k commits event e3;  
k gets data d3 from e3;  
...  
i sets restart;  
j sets purge;  
i unblocks with nonce1;  
j unblocks with nonce2;  
...

Monitor

loop(Map) →  
{Evt,Id,Nonce} = recv_event(),  
if Nonce ≠ null,  
    Map2 = Updt(Id,Nonce,Map);  
else  
    Map2 = Map;  
end,  
{PtrnMtch,AL,RL} = handle(Evt),  
if(PtrnMtch) →  
adpt(AL,Map2),  
release(RL,Map2),  
loop(Map2);  
end.

---
pre-instrumented synchronous adaptation functions:

- **silent-kill** (Terminations category – slnt\_kill(AL)\_RL): As outlined in Remark 4 (in the beginning of Sec. 4.1.1), this action instructs a blocked actor (using an adaptation request message) to willingly relinquish its supervision links prior to terminating itself. By doing so it refrains from alerting any of its supervisors or linked actors about its termination. This enables the monitor to terminate actors without possibly triggering any of the system’s built-in self-adaptation mechanisms. Unlike asynchronous killing, since the silent-kill requires the recipient actor to be blocked, it ensures that actors are terminated on-time, as blocking forbids an actor from progressing after committing an action.

- **purge** (Interception category – prg(AL)\_RL): Upon interpreting a purge request, the blocked actor uses a loop of non-blocking receives [7] to continuously read and ignore the messages in its mailbox. The loop stops when no more messages are left, i.e., the mailbox is emptied. In this case, synchronisation was essential given that the actor model strictly prohibits external actors (like our monitor) from accessing and modifying the mailbox contents of another actor. A variant of this adaptation is intercept, in which only the messages matching a specific pattern are purged from the mailbox, while the rest remain intact.

- **sync-link** (Interception category – sync-link(AL)\_RL): The monitor sends a link request message to an instrumented blocked system actor containing a list of process ids. When the instrumented code interprets this request, it links the host actor with all the process ids enlisted in the request message (if not already linked) in a timely manner. The same mechanism applies for the reverse function sync-unlink, which unlinks the instrumented actor from all the enlisted actors if a link exists.

- **restart** (Restarts category – restr(AL)\_RL): Restarting a process in Erlang is normally performed by first killing the process and then respawning it. However, since the adaptee may be linked to other actors, killing it might also terminate (or inform) other linked actors as a side-effect which could cause further complications. Moreover,
even if the actor is terminated silently with a silent-kill action, we still run into the following two problems:

– the links removed by the silent-kill must be relinked to the newly respawned process, in the same way as they were prior to killing the original actor;
– as respawning an adaptee actor changes its process id, we need to somehow inform the other actors about this change, otherwise they would not be able to contact the restarted process.

To avoid these problems with our restart implementation, a monitor restart request causes the blocked actor to: empty its mailbox (same as in purge), refresh its internal state (i.e., it removes any non-instrumentation data residing in its process dictionary) and finally restarts the actor by calling the original initial function used by the spawn operation that created the actor in question. In other words, the adaptee is not terminated and respawned; instead we call its initialisation function from within the actor’s own thread of control. This is better explained in Ex. 4.3.1.

**Example 4.3.1.** Recall the system given in Fig. 1.1, and assume that the common-interface i was at some point during system initialisation spawned by a parent process which called the function spawn(interface:start()). This spawn operation therefore creates a new actor process which internally calls the function interface:start(), i.e., the function which starts the common-interface from the very beginning. Whenever i is sent a restart request by our monitor, it empties its mailbox, refreshes its internal state and finally calls interface:start() to start afresh.

By using this mechanism, a process is able to retain its process identifier even after restarting. This is crucial to avoid having to inform other system actors about the new process id; failing to do so would lead to having other system actors unable to contact the restarted actor. Implementing this adaptation was not as straight forward as the previous ones, as this required a further extension to the AOP instrumentation framework. The extended AOP is now also able to save the details (module, function and arguments) of
the initial function used in spawn operation, within the internal state (i.e., its process dictionary) of the newly spawned actor. This information is then retrieved if and when the instrumented actor receives a restart adaptation request while it is blocked.

### 4.4 Strengthening Yaws through Runtime Adaptation

To provide a better insight on how runtime verification properties can easily be converted into runtime adaptation properties, we incrementally extend Yaws property (3.4) (given in Sec. 3.3.2). More specifically we strengthen the Yaws webserver with runtime adaptations for mitigating invalid HTTP requests that might contribute to a *dot-dot-slash* attack which exploits the directory traversal vulnerability. Hence we show that by introducing minimal constructs in the currently available RV scripts we can use our RA framework to mitigate third-party applications such as the Yaws webserver.

```plaintext
\[
\max X ( \\
[\text{acceptor}\?(hld, \text{next}.\_)] \\
[hld \models \text{ret}([\text{yaws}, \text{do_recv}, 3, \{\text{ok}, \{\text{http_req}, \text{GET}, \{\text{abs_path}, \text{path}, \_\}\}\}]]) \\
[hld \models \text{ret}([\text{yaws}, \text{do_recv}, 3, \{\text{ok}, \text{h1}\}])]
\ldots
[hld \models \text{ret}([\text{yaws}, \text{do_recv}, 3, \{\text{ok}, \text{h6}\}])]
[hld \models \text{ret}([\text{yaws}, \text{do_recv}, 3, \{\text{ok}, \text{http_eoh}\}])]
\text{if} \text{path} == \\
\text{"/pic.png" or else path} == \\
\text{"/site.html" then} \\
[hld \models \text{call}([\text{yaws_sendfile}, \text{send}, \_\_]])X \\
\text{else} \\
\text{slnt_kill(hld) prg(acceptor) X}
) \]
```

Contrasting to (3.4), instead of issuing a violation (ff), property (4.8) applies a *silent-kill* action (i.e., slnt_kill(..)) to immediately isolate and terminate the handler actor assigned to the requesting client, thus preventing the termination of the handler from possibly affecting other linked actors. Note that we are able to apply a silent-kill mitigation over the assigned handler actor (bound to hld), as our RA framework also supports applying adaptations over actors which are only known at runtime. This adaptation is then followed by a *purge* adaptation (i.e., prg(..)) that empties the mailbox of the acceptor.
As both the silent-kill and purge adaptations are synchronous adaptations (as explained in Sec. 4.3.1) we need to specify the necessary synchronisation operations. As these adaptations are applied on system actors $hId$ and acceptor, we must identify a set of necessities that must be converted into blocking necessities to ensure that the adaptations are applied correctly. The rule of thumb for identifying such candidates is that whenever we have an adaptation which applies on a system actor $i$, e.g., prg($i$), then a candidate necessity should:

(i) be defined prior to the adaptation;
(ii) have $i$ defined as its subject; and
(iii) be the closest candidate necessity lying in the path leading to the adaptation – this is important so that an actor $i$ is not blocked earlier than required.

For instance, to correctly apply the first slnt_kill($hId$) action declared in (4.8), we convert the end-of-headers necessity (i.e., $[hId\triangleright ret((yaws, do_recv, 3, \{ok, \text{http\_eoh}\})])$) into a blocking one. This candidate was chosen as it satisfies rules (i), (ii) and (iii). Using the same argument we add the blocking attribute to the first necessity, allowing the prg(acceptor) adaptation to be applied. The rest of the necessities are converted into asynchronous necessities. Hence by using these 3 rules we augment (4.8) with blocking necessities to obtain (4.9).

\[
\begin{align*}
\text{max } X \left( \\
\text{[acceptor?}\{hId, \text{next, }..\}\right)^b \\
[hId \triangleright ret((yaws, do_recv, 3, \{ok, \text{http\_eoh}\})])^a \\
[hId \triangleright ret((yaws, do_recv, 3, \{ok, h1\})])^a \\
\vdots \\
[hId \triangleright ret((yaws, do_recv, 3, \{ok, h6\})])^a \\
[hId \triangleright ret((yaws, do_recv, 3, \{ok, \text{http\_eoh}\})])^b \\
\text{if } \text{path} = "\text{/pic.png}" \text{ or else } \text{path} = "\text{/site.html}" \text{ then} \\
[hId \triangleright call((yaws\_sendfile, send, [.., \text{path, }.., ]))]^a X \\
\text{else} \\
\text{slnt\_kill(hId) prg(acceptor) } X \\
\end{align*}
\]

Thus far in (4.9), we have only added synchronisation to block the adaptees without ever releasing them. We must therefore release these blocked actors by specifying them
in the appropriate release-lists. As shown by (4.10) we add the acceptor to the release list of every necessity operation that lies between the blocking necessity which blocks the acceptor actor ([acceptor?\(hld, next, \_\)]\(^b\)), and the adaptation (i.e., prg(acceptor)). This ensures that the acceptor is released in case the property is trivially satisfied before reaching the point where the adaptation needs to be applied. Furthermore, we also add the handler and the acceptor to the release list of the last adaptation; this ensures that the actors are released after applying the adaptations.

\[
\max X \ ( \\
\text{[acceptor}!\{hld, next, \_\}]^b_x \\
[hld \Rightarrow \text{ret}((\text{yaws, do_recv, 3, [ok, \{http_req, \text{'GET'}, \{abs_path, path\}, \_\}]}))])^a_{\text{acceptor}} \\
\vdots \\
[hld \Rightarrow \text{ret}((\text{yaws, do_recv, 3, [ok, h6]}))]^a_{\text{acceptor}} \\
[hld \Rightarrow \text{ret}((\text{yaws, do_recv, 3, [ok, http_eoh]}))]^b_{\text{acceptor}} \\
\text{if } \text{path} == \text{"/pic.png" otherwise } \text{path} == \text{"/site.html" then} \\
[hld \Rightarrow \text{call}((\text{yaws_sendfile, send, [\_ path, \_ ]})])^a_{\text{acceptor}} X \\
\text{else} \\
\text{slnt_kill(hld)}_x \text{ prg(acceptor)}_{hld, acceptor} X
\]

(4.10)

Therefore the rule of thumb for releasing a blocked actor is to add its identifier to the release list of:

(i) every necessity lying between the point where it was blocked and its adaptation action; and to

(ii) the release list of the adaptation which is applied to it (or to those defined after it within the same path).

### 4.5 A Formal Model for Runtime Adaptation

In this section we develop a formal operational model of our runtime adaptation framework. The primary reasons for developing this formal model are:

- A formal model enables us to understand how the monitors specified by an arbitrary script are to behave at runtime without having to actually compile the script and run
the synthesised monitors. This also relieves us from having to deal with the complexities of the implementation.

- As during the course of this research we developed the model in parallel to the implementation, the model also served to guide the implementation (and vice-versa).
- The model allows us to prove that the operational behaviour of our framework corresponds to the semantics defined in earlier work [39].
- Formalisation also enables us to better understand the kind of errors that our RA scripts may introduce, thereby allowing us to look into static analysis techniques in Chapter 5 for detecting erroneous scripts prior to deployment.

We develop this model in two subsequent phases. As part of the first phase, in Sec. 4.5.1 we develop and present a preliminary operational model for detectEr’s original (unextended) logic (as presented in Fig. 2.10 in Sec. 2.4), given that currently such a model does not exist. We then prove that our operational model corresponds to detectEr’s semantics developed in previous work [39].

For the second phase, in Sec. 4.5.2 we formalise the mechanisms for applying adaptation and incremental synchronisation, in a way which reflects the implementation constraints (explained Sec. 4.1). We then augment these formalised mechanisms in our preliminary formal model.

### 4.5.1 An Operational Model for detectEr

A semantics [66] for the *closed* logic formulas (*i.e.*, no free term or formula variables) of the core logic presented in Sec. 2.4.3 is given as a Labelled Transition System (LTS). This is defined by the transition rules in Fig. 4.6. Note that for convenience we restate the logic’s syntax in Fig. 4.6 as well.

#### The Preliminaries

The LTS semantics models abstractly the monitoring of the respective property, and assumes: a set of (visible) actions $\alpha, \beta \in \text{Act}$, and a distinguished *silent* action, $\tau$; we let
4. Language Design, Implementation and Formalisation

Syntax
\[
\phi, \psi \in \text{Frm} ::= \text{tt} \mid \text{ff} \mid \phi \land \psi \mid [e]\phi \mid X \mid \text{max } X. \phi \mid \text{if } b \text{ then } \phi \text{ else } \psi
\]

Structural Equivalence Semantics
\[
\phi_1 \land \phi_2 \equiv \phi_2 \land \phi_1 \quad \phi_1 \land (\phi_2 \land \phi_3) \equiv (\phi_1 \land \phi_2) \land \phi_3 \\
\text{tt} \land \phi \equiv \phi \\
\text{ff} \land \phi \equiv \phi
\]

Reduction Semantics
\[
\begin{array}{c|c|c}
\text{rIDem1} & \text{rIDem2} \\
\hline
\text{ff} \xrightarrow{\alpha} \phi & \text{tt} \xrightarrow{\alpha} \phi
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{rTRU} & \text{rFLS} & \text{rSTR} \\
\hline
\text{b} \Downarrow \text{true} & \text{if } b \text{ then } \phi \text{ else } \psi \xrightarrow{\tau} \phi & \phi \equiv \phi' \xrightarrow{\gamma} \phi' \equiv \psi \\
\text{b} \Downarrow \text{false} & \text{if } b \text{ then } \phi \text{ else } \psi \xrightarrow{\tau} \psi & \phi \equiv \phi' \xrightarrow{\gamma} \phi' \equiv \psi
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{rCN1} & \text{rCN2} & \text{rCN3} \\
\hline
\phi \xrightarrow{\alpha} \phi' \quad \psi \xrightarrow{\alpha} \psi' & \phi \land \psi \xrightarrow{\tau} \phi' \land \psi' & \phi \land \psi \xrightarrow{\tau} \phi' \land \psi' \\
\phi \land \psi \xrightarrow{\alpha} \phi' \land \psi' & \phi \land \psi \xrightarrow{\tau} \phi' \land \psi' & \phi \land \psi \xrightarrow{\tau} \phi' \land \psi'
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{rMAX} & \text{rNC1} & \text{rNC2} \\
\hline
\text{max } X. \phi \xrightarrow{\tau} \phi[\text{max } X. \phi / X] & [e] \phi \xrightarrow{\alpha} \phi \sigma & \text{mtch}(e, \alpha) = \sigma \\
\text{max } X. \phi \xrightarrow{\tau} \phi[\text{max } X. \phi / X] & [e] \phi \xrightarrow{\alpha} \phi \sigma & \text{mtch}(e, \alpha) = \perp
\end{array}
\]

Figure 4.6: The Logic and its LTS Semantics

\gamma \text{ range over } \text{Act} \cup \{\tau\}. \text{ Visible actions represent system operations which may contain values } v, u \in \text{Val}, \text{ that range over either actor identifiers, } i, j, h \in \text{Pid}, \text{ or generic data } p \in \text{Data} \text{ such as integers } n, \text{ lists } p_1: \ldots: p_n, \text{ and tuples } \{p_1, \ldots, p_n\}. \text{ We also use metavariable } l \in (\text{Pid} \cup \text{Var}) \text{ to represent either a process identifier or a term variable.}

In this semantics we work up-to structural equivalence of formulas \(\phi \equiv \psi\) (see Equivalence Semantics in Fig. 4.6) for commutativity, associativity etc. The semantics also assumes a partial function \(\text{mtch}(e, \alpha)\), defined in Def. 4.5.1 for matching action patterns, \(e\), with visible system actions, \(\alpha\). When a match is successful, the function returns a substitution from the term variables found in the pattern, to the corresponding values of the matched action, \(\sigma::\text{Var} \rightarrow \text{Val}\).

Def. 4.5.1 (Pattern Matching). Function \(\text{mtch}(e, d)\) defines how a pattern \(e\) can be compared to another pattern \(d\). When a match occurs this function returns a mapping from term variables to actual values, while it returns \(\perp\) when the patterns do not match.
4. Language Design, Implementation and Formalisation

The `mtch(e, d)` function starts by checking whether the patterns define the same kind action e.g., whether both patterns define an output action. The `mtch` function also supports the universally matching pattern “⊥”, this matches any action pattern. If the action types match, this function uses the `pmtch(p', p'')` function to inspect the actions’ subjects and the data, where the latter includes values v, term variables x, tuples \( \{p'_1, \ldots, p'_n\} \) and lists \( p'_1; p'_2 \). The `pmtch(p', p'')` function checks that the data patterns correspond in both patterns e.g., a tuple of three matching entries are present in both patterns. If the data patterns match it returns a substitution environment \( \sigma \) which maps the term variables declared in one pattern, to the values of the other pattern.

Note that although the `mtch` function is defined for comparing necessity patterns i.e., `mtch(e, d)`, it can still be used to compare a necessity pattern \( e \) with a system action \( \alpha \) i.e., `mtch(e, \alpha)`, as well as to compare two system actions `mtch(\alpha, \beta)`. This is possible as system actions are equivalent to closed patterns i.e., patterns that do not define any
4. Language Design, Implementation and Formalisation

term variables. This pattern matching mechanism is further explained in the following example.

**Example 4.5.1.** We consider a number of scenarios where two patterns are pattern matched using the `mtch` function.

\[
\text{mtch}(\text{srv} \triangledown \text{clnt}!\{x, \text{ack}, y\}, \text{srv} \triangledown \text{clnt}!\{5, \text{ack}, z\}) = \{x \mapsto 5\} \quad (4.11)
\]
\[
\text{mtch}(\text{srv} \triangledown \text{clnt}!\{5, \text{ack}, z\}, \text{srv} \triangledown \text{clnt}!\{x, \text{ack}, y\}) = \{x \mapsto 5\} \quad (4.12)
\]

For instance in (4.11) the output (open) necessity pattern `srv \triangledown \text{clnt}!\{x, \text{ack}, y\}` is successfully matched with another (open) output pattern `srv \triangledown \text{clnt}!\{5, \text{ack}, z\}`. Note that term variable `x` is pattern matched with the value 5 and thus returned in the resultant substitution environment. Although variable `y` is also matched to variable `z`, the match function does not return an entry in the resultant substitution environment (remember that the substitution only maps variables to values). Also note that the match returns the same substitution environment if the arguments are inverted as shown in (4.12).

\[
\text{mtch}(\text{srv} \triangledown \text{clnt}!\{x, \text{ack}, y\}, \text{srv} \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}) = \{x \mapsto 5, y \mapsto \text{joe}\} \quad (4.13)
\]
\[
\text{mtch}(\text{srv} \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}, \text{srv} \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}) = \{\} \quad (4.14)
\]

In (4.13) the necessity pattern `srv \triangledown \text{clnt}!\{x, \text{ack}, y\}` is now matched with a closed pattern i.e., a system action `srv \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}`, where `x` and `y` are pattern matched with the values 5 and `\text{joe}` respectively. Moreover, in (4.14) the two closed actions are matched (exactly), thereby returning the empty substitution.

\[
\text{mtch}(\text{clnt} \triangledown \text{srv}!\{x, \text{ack}, y\}, \text{srv} \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}) = \bot \quad (4.15)
\]
\[
\text{mtch}(\text{clnt}?\{x, \text{ack}, y\}, \text{srv} \triangledown \text{clnt}!\{5, \text{ack}, \text{joe}\}) = \bot \quad (4.16)
\]

The mismatch in (4.15) is caused by the mismatching subjects of the output actions i.e., `srv` versus `clnt`, whereas the mismatch in (4.16) is caused since the different type of actions cannot pattern match, i.e., input action `clnt?\{x, \text{ack}, y\}` and output action `srv!\text{clnt}\{5, \text{ack}, \text{joe}\}` do not match.
The Modelled Reductions

In Fig. 4.6, we give two kinds of reductions namely $\alpha$-reductions and $\tau$-reductions, which are respectively represented using judgements $\varphi \xrightarrow{\alpha} \varphi'$ and $\varphi \xrightarrow{\tau} \varphi'$. The first judgement, i.e., $\varphi \xrightarrow{\alpha} \varphi'$, represents cases where a monitor $\varphi$ evolves (progresses) into a different monitor $\varphi'$ after observing an external system event $\alpha$. This models the implementation mechanism where a monitor reads a system event from its mailbox and progresses accordingly. The second judgement, i.e., $\varphi \xrightarrow{\tau} \varphi'$, models an internal (silent) action that a monitor $\varphi$ can perform so to evolve into $\varphi'$. Internal actions include operations such as unfolding recursive definitions and evaluating if-statement conditions. Using these judgements we can represent a monitor execution as a sequence of internal ($\tau$) and external ($\alpha$) reductions as shown in Ex. 4.5.2.

**Example 4.5.2.** Reduction sequence (4.17) models the execution of a monitor $\psi$ wrt. the following sequence of observable system events: $\alpha_1; \ldots; \alpha_n$.

$$\psi (\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^* \psi'$$

(4.17)

As shown by (4.17), a monitoring reduction over an observation $\alpha_i$ may be interposed by internal (silent) $\tau$-reductions, i.e., $\psi''(\xrightarrow{\tau})^* \xrightarrow{\alpha_i} (\xrightarrow{\tau})^* \psi'''$.

More specifically, in Fig. 4.6 formulas $tt$ and $ff$ are idempotent wrt. external transitions and interpreted as final states (verdicts). Conditional formulas silently (internally) branch to the respective subformula depending on the evaluation of a decidable boolean expression ($r\text{Tru}$ and $r\text{Fls}$), whereas rule $r\text{Max}$ internally unfolds a recursive formula. Necessity formulas, $[e]\varphi$, transition only with a visible action, $\alpha$. If the action $\alpha$ matches the necessity’s pattern (i.e., $\text{m}tch(e, \alpha) = \sigma$), it transitions to the continuation subformula $\varphi$ where the variables bound by the matched pattern are substituted with the respective matched values obtained from the action, i.e., it reduces into $\varphi \sigma$ where $\sigma$ contains the data mappings. Otherwise, the necessity formula transitions to $tt$ in case of a mismatch ($r\text{Nc2}$).

The rules for conjunction formulas model the parallel execution of subformulas as described in [39], whereby subformulas are allowed to perform independent silent $\tau$-
transitions (\(rCn2\) and \(rCn3\)) but transition together for external (\(\alpha\)) actions depending on their individual transitions (\(rCn1\)). Finally, \(rS\) allows us to abstract over structurally equivalent formulas. We write \(\varphi \xrightarrow{r} \psi\) in lieu of \(\varphi(\xrightarrow{r})^* \xrightarrow{r} (\xrightarrow{r})^* \psi\). We let \(t \in \text{Act}^*\) range over lists of visible actions and write \(\varphi \xrightarrow{t} \psi\) to denote \(\varphi \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \psi\) where \(t = \alpha_1 \ldots \alpha_n\).

**Example 4.5.3.** Recall property \(\varphi\) from (2.1) of Ex. 2.4.1 (restated below).

\[
\varphi \overset{\text{def}}{=} \max Y. [i?\{\text{inc}, x, y\} \left( ([j?y!\{\text{res}, x + 1\}] Y) \& ([\_\_\_y!\text{err}] \text{ff}) \right)]
\]

Using the semantics of Fig. 4.6 we can express how formula \(\varphi\) reduces to a violation when executed wrt. to the following trace:

\(i?\{\text{inc}, 5, h\}; j?h!\{\text{res}, 6\}; i?\{\text{inc}, 3, h'\}; k?h'!\text{err}\)

The runtime derivation is given as follows:

\[
\begin{align*}
\varphi & \xrightarrow{r\text{Max} \text{and } r\text{Nc1}} ([j?y!\{\text{res}, x + 1\}] \varphi) \& ([\_\_\_y!\text{err}] \text{ff}) [x\mapsto 5, y\mapsto h] \\
& \equiv ([j?h!\{\text{res}, 5 + 1\}] \varphi) \& ([\_\_\_h!\text{err}] \text{ff})
\end{align*}
\]

(\(\varphi\) reduced with rules \(r\text{Max}\) and \(r\text{Nc1}\), where \(\text{mtch}(i?\{\text{inc}, x, y\}, i?\{\text{inc}, 5, h\}) = \{x\mapsto 5, y\mapsto h\}\))

\[
\begin{align*}
\varphi & \xrightarrow{j?\{\text{inc}, 3, h'\}} ([j?h'!\{\text{res}, 3 + 1\}] \varphi) \& ([\_\_\_h'!\text{err}] \text{ff}) \\
& \xrightarrow{k?h'!\text{err}} \text{ff} \quad \blacksquare
\end{align*}
\]

**The Correspondence Evaluation**

Although we have developed LTS semantics which describe the runtime behaviour of the formulas expressed in detectEr’s logic, we still do not have any guarantees that this semantics corresponds to other semantics already available for detectEr [39].

As outlined in Sec. 2.4 in [39] the authors provide satisfaction semantics describing how an actor system satisfies a formal property, along with violation semantics which describe how a finite trace of system events, violates a formal property expressed in detectEr’s logic.

Similar to the way that the authors of [39] prove semantic correspondence between the satisfaction and violation semantics, in Thm. 4.5.1 we prove that our LTS semantics
corresponds to the violation semantics. The primary reason for proving correspondence wrt. the violations semantics (and not the satisfaction semantics), is that violations are defined wrt. a finite trace \( t \), which in some sense is analogous to the way a formula \( \varphi \) reduces into \( \varphi' \) over a trace \( t \), i.e., \( \varphi \Rightarrow \varphi' \).

**Theorem 4.5.1 (Semantic Correspondence).**

\[
A, t \models_v \varphi \iff (\varphi \Rightarrow \text{ff} \text{ and } A \overset{\gamma}{\Rightarrow})
\]

In Thm. 4.5.1 we prove that if an arbitrary actor system \( A \) generates a trace \( t \) which violates \( \varphi \), i.e., \( A, t \models_v \varphi \), then for the same trace \( t \), the formula \( \varphi \) should reduce to \( \text{ff} \) (i.e., to a violation) with our LTS semantics. We also prove the converse, i.e., if \( \varphi \) reduces to \( \text{ff} \) wrt. some trace \( t \) with our LTS semantics, then the respective violation relation should also hold.

We here give an outline of the main cases of the proof for Thm. 4.5.1. The reader may safely skip the details of the proof and proceed with Sec. 4.5.2.

**Proof.** During this proof we make reference to Lemma C.1.1 (stated below).

**Lemma C.1.1.**

\[
\varphi \overset{\gamma}{\Rightarrow} \varphi' \text{ and } B, t \models_v \varphi' \text{ and } A \overset{\gamma}{\Rightarrow} B \text{ implies } A, \gamma t \models_v \varphi
\]

This Lemma is proved by rule induction on \( \varphi \overset{\gamma}{\Rightarrow} \varphi' \). We provide a proof outline of this lemma in Appendix Sec. C.1.

We prove the only-if case for Thm. 4.5.1 by rule induction on \( A, t \models_v \varphi \). We here outline the main cases.

**Case** \( A, t \models_v [e] \varphi \): We know

\[
t = \alpha t' \tag{4.18}
\]
\[
A \overset{\alpha}{\Rightarrow} B \tag{4.19}
\]
\[
\text{mtch}(e, \alpha) = \sigma \tag{4.20}
\]
\[
B, t' \models_v \varphi \sigma \tag{4.21}
\]

By (4.21) and IH we know

\[
\varphi \sigma \overset{\gamma}{\Rightarrow} \text{ff} \tag{4.22}
\]
\[
B \overset{\gamma}{\Rightarrow} \tag{4.23}
\]
Thus, by (4.19) and (4.23) we obtain $A \overset{\alpha'}{\Rightarrow}$, i.e., $A \overset{\tau}{\Rightarrow}$ by (4.18), and by (4.20), (4.22) and rNc1 we derive $[e] \varphi \overset{\alpha'}{\Rightarrow} \text{ff}$, and by (4.18) this means $[e] \varphi \overset{\tau}{\Rightarrow} \text{ff}$ as required.

**Case $A, t \not\vdash \text{max } X. \varphi$:** We know

$$A, t \not\vdash \varphi[X/\text{max } X. \varphi]$$  \hspace{1cm} (4.24)

By (4.24) and IH we know

$$\varphi[X/\text{max } X. \varphi] \overset{\tau}{\Rightarrow} \text{ff}$$  \hspace{1cm} (4.25)

$$A \overset{\tau}{\Rightarrow}$$  \hspace{1cm} (4.26)

By rMax we know $\text{max } X. \varphi \overset{\tau}{\Rightarrow} \varphi[X/\text{max } X. \varphi]$, and hence, by (4.25), we obtain $\text{max } X. \varphi \overset{\tau}{\Rightarrow} \text{ff}$ as required.

**Case $A, t \not\vdash \varphi \& \psi$:** We know

$$A, t \not\vdash \varphi$$  \hspace{1cm} (4.27)

or

$$A, t \not\vdash \psi$$  \hspace{1cm} (4.28)

We must consider the following cases:

**Subcase $t = \varepsilon$:** By (4.27), (4.28) and since $t = \varepsilon$ we know

$$\varphi = \text{ff}$$  \hspace{1cm} (4.29)

or

$$\psi = \text{ff}$$  \hspace{1cm} (4.30)

By (4.29) and since $\text{ff} \& \psi \equiv \text{ff}$ we know $\text{ff} \& \psi \overset{\varepsilon}{\Rightarrow} \text{ff}$ as required. Similarly, by (4.30) and since $\varphi \& \text{ff} \equiv \text{ff}$ we know $\varphi \& \text{ff} \overset{\varepsilon}{\Rightarrow} \text{ff}$.

**Subcase $t = \alpha t$:** Since $t = \alpha t$, by (4.27) and IH we know

$$\varphi \overset{\alpha t}{\Rightarrow} \text{ff}$$  \hspace{1cm} (4.31)

$$A \overset{\alpha t}{\Rightarrow}$$  \hspace{1cm} (4.32)

Similarly, since $t = \alpha t$, by (4.28) and IH we know

$$\varphi \overset{\alpha t}{\Rightarrow} \text{ff}$$  \hspace{1cm} (4.33)

$$A \overset{\alpha t}{\Rightarrow}$$  \hspace{1cm} (4.34)

From (4.31) and (4.33) we can deduce

$$\varphi \overset{\alpha}{\Rightarrow} \varphi'$$  \hspace{1cm} (4.35)

$$\varphi' \overset{\tau}{\Rightarrow} \text{ff}$$  \hspace{1cm} (4.36)

$$\psi \overset{\alpha}{\Rightarrow} \psi'$$  \hspace{1cm} (4.37)

$$\psi' \overset{\tau}{\Rightarrow} \text{ff}$$  \hspace{1cm} (4.38)
By (4.35), (4.37) and rCn1 we know
\[ \varphi \& \psi \xrightarrow{\alpha} \varphi' \& \psi' \] (4.39)

By (4.36), (4.38) and (4.39) we know
\[ \varphi \& \psi \xrightarrow{\alpha} \varphi' \& \psi' \quad \text{if } \& \text{ ff } \equiv \varphi \& \psi \xrightarrow{\alpha} \text{ ff} \] (4.40)

Hence this subcase holds by (4.34) and (4.40).

Subcase \( t = \tau \) (\( \varphi \) makes the \( \tau \) transition): Since \( t = \tau \), by (4.28) and IH we know
\[ \varphi \xrightarrow{\tau} \text{ ff} \] (4.41)
\[ A \xrightarrow{\tau} \equiv A \xrightarrow{\tau} \] (4.42)

From (4.41) we can deduce
\[ \varphi \xrightarrow{\tau} \varphi' \] (4.43)
\[ \varphi' \xrightarrow{\tau} \text{ ff} \] (4.44)

By (4.43) and rCn2 we know
\[ \varphi \& \psi \xrightarrow{\tau} \varphi' \& \psi \] (4.45)

We know that for trace \( t, \psi \) reduces to some \( \psi' \) such that by (4.45) we can conclude
\[ \varphi \& \psi \xrightarrow{\tau} \varphi' \& \psi' \quad \text{ff } \& \psi' \equiv \varphi \& \psi \xrightarrow{\tau} \text{ ff} \] (4.46)

Hence this subcase holds by (4.42) and (4.46).

Subcase \( t = \tau \) (\( \psi \) makes the \( \tau \) transition):

The proof for this subcase is synonymous to that of the previous subcase.

We prove the if case by numerical induction on the number of transitions used in \( \varphi \xrightarrow{\tau} \text{ ff} \).

\( n = 0 \): We know that \( \varphi = \text{ ff} \) and \( t = \varepsilon \) and we trivially have \( A, \varepsilon \vDash \varphi \) ff for arbitrary \( A \).

\( n = k + 1 \): For some \( \varphi' \) we know \( t = \gamma t' \) such that
\[ \varphi \xrightarrow{\gamma} \varphi' \] (4.47)
\[ \varphi' \xrightarrow{\tau'} \text{ ff} \quad \text{where } |\tau'| = k. \] (4.48)

Similarly we also know
\[ A \xrightarrow{\gamma} B \] (4.49)
\[ B \xrightarrow{\tau'} \] (4.50)

By (4.48), (4.50) and IH we know
\[ B, \tau' \vDash \varphi' \] (4.51)

By (4.47), (4.49), (4.51) and Lemma C.1.1 we know \( A, \gamma t' \vDash \varphi' \) as required. □
4.5.2 Extending the Operational Model with Runtime Adaptation

Fig. 4.7 (below) describes a semantics for the extended logic with adaptations as discussed in Sec. 4.1 to 4.3. Recall that in Table 4.1 (given in Sec. 4.1.2) we distinguished between synchronous and asynchronous classes of adaptations. Given that adaptations pertaining to the same class are applied (to the adaptee) in the same manner, in this extended logic we work at this level of abstraction.

We therefore represent asynchronous and synchronous classes of adaptations as $aA(w)$ and $sA(w)$, respectively: they both take two lists of actor references as argument — $w, r \in (Pid \cup Var)^*$ — which are synonymous to the adaptation and release lists. Furthermore, the extended logic also includes the extended necessity, $[e]_w^\varphi$, where in this case $w$ is analogous to the release list, $RL$, of the necessity.

The extended logic also uses two additional constructs, $blk(r)\varphi$ and $rel(r)\varphi$; these are not meant to be part of the specification scripts, but are used as part of the runtime syntax. Since the extended logic can also affect the system being monitored, through adaptations and synchronisations, the operational semantics is given in terms of configurations, $s \triangleright \varphi$, where $s$ is an abstract representation of the system and $\varphi$ is a closed formula. Note that in the previous operational model (given in Fig. 4.6), giving an abstract representation of the system was not necessary as the formulas were passive, i.e., unable to effect the system. By contrast in the extended logic we now include adaptations and synchronisation mechanisms that can effect the system; an abstract representation $s$ therefore allows us to model some of these effects at a high level of abstraction.

In addition to closed formulas $\varphi$, configurations thus include the monitored system represented abstractly as a partial map, $s : Pid \rightarrow \{\bullet, \circ\}$, describing whether an actor (through its unique identifier) is currently blocked (suspended), $\bullet$, or executing, $\circ$. We occasionally write $w : \bullet$ to denote the list of mappings $i_1 : \bullet, \ldots, i_n : \bullet$ where $w = i_1, \ldots, i_n$ (similarly for $w : \circ$).

In the operational model given in Fig. 4.7 we also distinguish between two kinds of reductions that can be carried out by a configuration, namely $\alpha$-reductions, formalised as $s \triangleright \varphi \xrightarrow{\alpha} s' \triangleright \varphi'$, and $\tau$-reductions i.e., $s \triangleright \varphi \xrightarrow{\tau} s' \triangleright \varphi'$. The former defines cases where
4. Language Design, Implementation and Formalisation

Extended Logic Syntax with Adaptations and Synchronisations

\[ \varphi, \psi \in \text{Frm} ::= \ldots | [e]_w \varphi | \text{aA}(r)_w \varphi | \text{sA}(r)_w \varphi | \text{blk}(r) \varphi | \text{rel}(r) \varphi \]

Monitor Transition Rules

\[ \begin{align*}
\text{rNc1} & : \text{mtch}(e, \alpha) = \sigma & \text{subj}(\alpha) = i & [e]_w \alpha \rightarrow \text{blk}(i) (\varphi \sigma) \\
\text{rNc2} & : \text{mtch}(e, \alpha) = \sigma & [e]_w \alpha \rightarrow \varphi \sigma \\
\text{rNc3} & : \text{mtch}(e, \alpha) = \bot & [e]_w \alpha \rightarrow \text{rel}(r) \varphi \\
\text{rAdA} & : \text{aA}(w), \varphi \rightarrow \text{rel}(r) \varphi \\
\text{rAdS} & : \text{sA}(w), \varphi \rightarrow \text{rel}(r) \varphi \\
\text{rRel} & : \text{rel}(r) \varphi \rightarrow \varphi \\
\text{rBLK} & : \text{blk}(r) \varphi \rightarrow \varphi \\
\text{rCN4} & : \varphi \rightarrow \varphi' \\
\end{align*} \]

System Transition Rules

\[ \begin{align*}
\text{sNEW} & : s \tau \rightarrow s, i: \circ \\
\text{sAct} & : \text{subj}(\alpha) = i & \text{ids}(\alpha) \subseteq \text{dom}(s) & s, i: \circ \alpha \rightarrow s, i: \circ \\
\text{sBLK} & : \text{b}(w) & s, w: \circ \rightarrow s, w: \bullet \\
\text{sREL} & : \text{r}(w) & s, w: \bullet \rightarrow s, w: \circ \\
\text{sAdA} & : w \subseteq \text{dom}(s) & s \alpha \rightarrow s \\
\text{sAdS} & : s, w: \circ \rightarrow s, w: \bullet \\
\end{align*} \]

Instrumentation Transition Rules

\[ \begin{align*}
\text{iAdA} & : \varphi \rightarrow \varphi' & s \rightarrow \varphi' & s' \rightarrow \varphi' \\
\text{iAct} & : \varphi \rightarrow \varphi' & s \alpha \rightarrow s' \varphi \rightarrow \varphi' \\
\text{iTrM} & : \varphi \rightarrow \varphi' & s \rightarrow \varphi' & s' \rightarrow \varphi' \\
\text{iSys} & : s \rightarrow \varphi \rightarrow s' \rightarrow \varphi \\
\text{iMon} & : s \rightarrow \varphi \rightarrow s \rightarrow \varphi' \\
\end{align*} \]

Figure 4.7: A Runtime Semantics for Instrumented Properties with Adaptations

A system \( s \) commits an external monitorable event \( \alpha \) which is perceived by monitor \( \varphi \); due to this event, both the system and the monitor may evolve together thus forming a new configuration \( s' \triangleright \varphi' \). The latter (i.e., \( \tau \)-reductions) formalise cases where the whole configuration \( s \triangleright \varphi \) performs an internal transition into another configuration \( s' \triangleright \varphi' \) where \( s' \) and \( \varphi' \) are the evolved versions of \( s \) and \( \varphi \) respectively. These internal transitions can be caused in the following cases:

- System \( s \) performs an internal operation which is unobservable by \( \varphi \), and thus \( s \)
evolves into some $s'$ independently of formula $\varphi$.

- Formula $\varphi$ performs the internal transition thereby reducing into some formula $\varphi'$, independently from the system $s$.

- System $s$ and formula $\varphi$ interact over a synchronisation operation or an adaptation action – this causes both $s$ and $\varphi$ to evolve together into some new $s'$ and $\varphi'$, according to the exchanged interaction.

More specifically, to describe adaptation interactions between the monitor and the system, the LTS semantics of Fig. 4.7 employs four additional labels, ranged over by the variable $\mu$. These include the asynchronous and synchronous adaptation labels, $a(w)$ and $s(w)$, to denote respectively that an asynchronous and synchronous action affecting actors $w$, has been executed. They also include a blocking action, $b(w)$, and an unblocking (release) action, $r(w)$, affecting the execution of actors with identifiers in the list $w$.

The semantics is defined in terms of three LTSs: one for logical formulas (i.e., monitors), one for systems, and one for configurations which is based on the other two LTSs.

**The Monitor Transition Rules**

These extend the rules in Fig. 4.6 with the exception of those for the necessity formulas, which are replaced by rules $r_{Nc1}$, $r_{Nc2}$ and $r_{Nc3}$. Whereas the new $r_{Nc2}$ follows the same format as that of $r_{Nc1}$ from Fig. 4.6, the rule for blocking necessity formulas, $r_{Nc1}$, transitions into a blocking construct, $blk(i) \varphi$, for blocking the subject of the action (i.e., actor $i$) in case a pattern match is successful. Note that the action’s subject is retrieved using the $\text{subj}(e)$ function as defined in Def. 4.5.2 (below). Furthermore, in case an action mismatches a necessity pattern, $r_{Nc3}$ transitions the necessity formula to a release construct, $rel(r) \varphi$, with the specified release list of actors, $r$.

**Def. 4.5.2 (The subject Function).**

$$\text{subj}(\gamma) \equiv \begin{cases} l_1 & \text{if } e = l_1 \triangleright l_2 \triangleright ! p \\ l & \text{if } e = l ? p \\ l & \text{if } e = l \triangleright \text{call}(p) \\ l & \text{if } e = l \triangleright \text{ret}(p) \end{cases}$$
This function inspects a necessity pattern $e$ and extracts the process identifier of the actor that committed the action. In particular note that in output actions e.g., $i \rightarrow j 13$, the subject is $i$, as this actor is sending the message to $j$, and hence it is the one performing the output action.

Asynchronous and synchronous adaptation formulas, $aA(w),\varphi$ and $sA(w),\varphi$, transition with the respective labels (i.e., $a(w)$ and $s(w)$), using rules $rAdA$ and $rAdS$, to a release construct, $\text{rel}(r)$, thereby communicating the respective adaptations to $w$. Similarly the block and release constructs reduce over labels (i.e., $b(r)$ and $r(r)$ with rules $rRel$ and $rBlk$ respectively. Finally, rule $rCn4$ allows monitor actions affecting the system, i.e., $\mu$, to be carried out under a conjunction formula, independent of the other branch; we elide the obvious symmetric rule $rCn5$.

**The System Transition Rules**

These rules allow further actor spawning (rule sNew in Fig. 4.7) but restrict external actions ($\alpha$) to those whose subject is currently active, i.e., unblocked $i : \circ$ (rule sAct); external actions model the system events that can be reported to the monitor. Whereas asynchronous adaptations can be applied to any actor list, irrespective of their status i.e., can be either $w : \circ$ or $w : \bullet$ (rule sAdA), synchronous ones require the adaptees to be blocked i.e., must be $w : \bullet$ (rule sAdS). This transpires from our implementation constraints where synchronous adaptations require the respective system actors to be blocked by our instrumented code in some prior stage, so that the monitor can later on communicate synchronous adaptations as messages, which are then received and interpreted by the instrumented actor. By contrast, this is not required by asynchronous adaptations.

Finally rules $sB\text{l}k$ and $sR\text{el}$ model the blocking and releasing mechanism used by the RA protocol code instrumented in the system. In fact, blocking is modelled in terms of a transition which changes the status of the respective adaptees from active $i : \circ$, to blocked $i : \bullet$; the converse applies for releasing.
The Instrumentation Rules

The *instrumentation rules* for configurations describe how system (visible) actions, $\alpha$, affect the monitors, and dually, how the monitor adaptation and synchronisation actions, $\mu$, affect the system. For instance, if the monitor instigates action $\mu$ and the system allows it, they both transition together as a silent (internal) transition (rule $i\text{Ada}$). Dually, if the system generates external action $\alpha$ and the monitor can observe it, they also transition together (rule $i\text{Act}$); this essentially models the event reporting mechanism whereby monitorable events are forwarded to the monitor by the system.

If the monitor cannot observe an external $\alpha$-action (rule $i\text{Trm}$), it terminates as formula $\top$. Importantly note that both rules $i\text{Act}$ and $i\text{Trm}$ require the monitor not to be in a position to perform an adaptation/synchronisation action, i.e., premise $\varphi \not\mu^{-}\rightarrow$; this gives precedence to monitor actions over system ones in our instrumentation. This precedence models the fact that in our RA protocol implementation (refer to Sec. 4.3), the monitoring loop only perceives a system event by reading it from its mailbox after it is done applying the necessary handling actions, which may include $\mu$-actions e.g., issuing an adaptation. Rules $i\text{Sys}$ and $i\text{Mon}$ allow systems and monitors to perform internal transitions (wrt. $\tau$-actions) independently of each other.

**Remark 7.** We point out the subtle difference between rules $i\text{Act}$ and $i\text{Ada}$. Rule $i\text{Act}$ states that whenever the system $s$ performs an external $\alpha$-action which is perceived by formula $\varphi$, then the *entire configuration* $s \varphi$ transitions over the external action, i.e., $s \varphi \overset{\alpha}{\rightarrow} s' \varphi'$. This opposes rule $i\text{Ada}$ in which whenever the system $s$ and formula $\varphi$ interact over a synchronisation or adaptation action $\mu$, then the configuration transitions silently, i.e., $s \varphi \overset{\tau}{\rightarrow} s' \varphi'$ instead of $s \varphi \overset{\mu}{\rightarrow} s' \varphi'$. This transpires since external $\alpha$-actions may be observable by other external systems which interact with the monitored system, and hence these actions must be kept observable. Contrasting this, $\mu$-actions model interactions that are only relevant (i.e., internal) to the system and monitor, and hence they are not be made perceivable to external systems.
4. Language Design, Implementation and Formalisation

**Formalising Synchronisation Errors**

In the following example we use our extended operational model to demonstrate how adaption formulas reduce at runtime when executed \( w.r.t. \) a system. More importantly we show the subtle errors that erroneously specified synchronisation operations may introduce at runtime.

**Example 4.5.4.** Recall the adaptation formula \( \varphi' \) defined in (4.7) of Ex. 4.2.2 (restated below).

\[
\varphi' \overset{\text{def}}{=} \max Y. [i?\{\text{inc}, x, y\}]^a_e \hspace{1cm} (\hspace{1cm} [j\triangleright y\{\text{res}, x + 1\}]^b_e Y) \hspace{1cm} \& \hspace{1cm} ([z\triangleright y\{\text{err}\}]^b_i \text{restr}(i)_e \text{prg}(z)_{i\cdot\varphi'})
\]

Now, consider the following system \( s = ((i, j, k, h) : \circ) \) where all of the system actors (i.e., \( i, j, k \) and \( h \)) are unblocked. We can therefore model the runtime execution of formula \( \varphi' \) \( w.r.t. \) system \( s \) with adaptations as follows:

\[
s \triangleright \varphi' \overset{i?\{\text{inc}, 1, h\}}{\longrightarrow} \overset{i+k?\{\text{inc}, 1, h\}}{\longrightarrow} s \triangleright \text{blk}(i) \left(\begin{array}{c}
[j\triangleright h\{\text{res}, 2\}]^a_e \varphi' \hspace{1cm} \& \hspace{1cm} [z\triangleright h\{\text{err}\}]^b_i \text{restr}(i)_e \text{prg}(z)_{i\cdot\varphi'}
\end{array}\right) \quad (4.52)
\]

\[
\tau \rightarrow ((j, h, k) : \circ, i : \bullet) \triangleright \left(\begin{array}{c}
[j\triangleright h\{\text{res}, 2\}]^a_e \varphi' \hspace{1cm} \& \hspace{1cm} [z\triangleright h\{\text{err}\}]^b_i \text{restr}(i)_e \text{prg}(z)_{i\cdot\varphi'}
\end{array}\right)
\]

\[
\overset{k+h!\text{err}}{\longrightarrow} ((j, h, k) : \circ, i : \bullet) \triangleright \text{blk}(k) \text{restr}(i)_e \text{prg}(k)_{i,k} \varphi' \quad (4.53)
\]

\[
\tau \rightarrow ((j, h) : \circ, i : \bullet, k : \bullet) \triangleright \text{restr}(i)_e \text{prg}(k)_{i,k} \varphi' \quad (4.54)
\]

\[
\tau \rightarrow ((j, h) : \circ, i : \bullet, k : \bullet) \triangleright \text{rel}(i, k) \varphi' \quad \overset{\tau}{\longrightarrow} s \triangleright \varphi' \quad (4.55)
\]

In particular, the blocking necessities matched in (4.52) and (4.54) yield the actor blocking constructs which block actors \( i \) and \( k \) in (4.53) and (4.55) respectively. This allows for the synchronous adaptations in (4.56) to be performed, thereby proceeding to release of the respective actors.

Erroneous blocking directives however may lead to stuck synchronous adaptations i.e., rule sAnS is inapplicable on the system. For instance, if we change the first blocking
necessity in \( \varphi' \) of (4.7) to an asynchronous one, \([i ? \text{inc}, x, y] \)^{a} \( \text{prg} \), it yields the execution below, where \( \varphi'' \) is the erroneous formula:

\[
\varphi'' \overset{\text{def}}{=} \max Y. [i ? \text{inc}, x, y]_{x}^{a} [i ! \text{inc}, x, y]_{x}^{b} \left( \left( [j \triangleright y \text{prg}, x + 1]^{a}_{x} Y \right) \& \left( [z \triangleright y \text{err}]^{b}_{z} \text{prg} \right) \right)
\]

The final configuration in (4.57) (i.e., \( ((i, j, h) : o, k : \bullet) \triangleright \text{prg}(i) \_k \varphi'' \)) is stuck because the synchronous adaptation on \( i \) (i.e., \( \text{prg}(i) \)) cannot be carried out since \( i \) is not blocked (i.e., since \( i : o \)). A similar situation is reached if a blocked actor is released prematurely by a concurrent branch. For instance, if we erroneously change the release list of the necessity subformula \([j \triangleright y \text{prg}, x + 1]^{a}_{x} \) from \( \varepsilon \) to \( i \), this releases \( i \) upon mismatch, interfering with adaptation actions along the other branch of the conjunction. Hence this erroneous formula (\( \varphi'' \)) exhibits a race condition whereby it can non-deterministically yield a number of executions, some of which may lead to stuck configurations. Here we present two possible executions, where one of them applies the \( \text{prg}(i) \) adaptation erroneously:

\[
\varphi'' \overset{\text{def}}{=} \max Y. [i ? \text{inc}, x, y]_{x}^{a} [i ! \text{inc}, x, y]_{x}^{b} \left( \left( [j \triangleright y \text{prg}, x + 1]^{a}_{x} Y \right) \& \left( [z \triangleright y \text{err}]^{b}_{z} \text{prg} \right) \right)
\]

**Execution 1:** The adaptation is erroneously applied.

The concurrent branch \( \text{rel}(i) \_t \) in (4.58) is evaluated first, thereby releasing actor \( i \) prior to applying the restart action \( \text{prg}(i) \) defined in the opposing branch (i.e., \( \text{blk}(k) \_k \text{prg} \_k \dot{\text{...}} \)). This leads to the stuck configuration shown in (4.59) where adaptation \( \text{prg}(i) \) cannot be applied (with rule 1Ada) as actor \( i \) is active, i.e., \( i : o \).

\[
\Rightarrow ((i, j, h) : o, k : \bullet) \triangleright \text{prg}(i) \_k \varphi''
\]
Execution 2: The adaptation is validly applied.

The adaptation can however be validly applied in case the concurrent branch in (4.58) containing the restart adaptation (i.e., branch blk(k) restr(i)...) is evaluated before the releasing branch as shown in (4.60) below.

\[
\tau \rightarrow ((j, h) : o, (i, k) : \bullet) \triangleright (\text{rel}(i) \ tt) \text{ & } (\text{prg}(k))_{i,k} \varphi''')
\]

\[
\tau \rightarrow ((j, h) : o, (i, k) : \bullet) \triangleright (\text{rel}(i) \ tt) \text{ & } (\text{prg}(k))_{i,k} \varphi''')
\]

(4.60)

Synchronisation errors caused due to concurrency can be more difficult for the specifier to notice without any assistance. Similarly, synchronisation errors can also be introduced in cases where the monitor attempts to release an unblocked actor. For instance, if we continue with the second execution (i.e., from (4.60)), the configuration progresses to a point where both concurrent branches attempt to release actor \(i\). In the derivation below we consider two possible continuations, starting from (4.60), both of which lead to a stuck configuration.

Continuation 1:

\[
\tau \rightarrow ((i, j, h) : o, k : \bullet) \triangleright \varphi''')
\]

\[
\tau \rightarrow ((i, j, h) : o, k : \bullet) \triangleright \varphi''')
\]

\[
\tau \rightarrow ((i, j, h, k) : o) \triangleright \varphi''')
\]

(4.61)

Continuation 2:

\[
\tau \rightarrow ((j, h) : o, (i, k) : \bullet) \triangleright (\text{rel}(i) \ tt) \text{ & } (\text{prg}(k))_{i,k} \varphi''')
\]

\[
\tau \rightarrow ((j, h) : o) \triangleright (\text{rel}(i) \ tt) \text{ & } \varphi''')
\]

(4.62)

Once again in both (4.61) and (4.62) we have reached a stuck configuration because a second release action on \(i\) could not be carried out as this was already released.

The semantics of Fig. 4.7 allows us to formalise configurations in an erroneous state (as identified in Ex. 4.5.4) i.e., when a monitor wants to release or apply synchronous adaptations that the system prohibits, as follows:

Def. 4.5.3. \(\text{error}(s \triangleright \varphi) \overset{\text{def}}{=} (\varphi \overset{s(w)}{\longrightarrow} \text{ and } s \overset{s(y)}{\longrightarrow}) \text{ or } (\varphi \overset{r(w)}{\longrightarrow} \text{ and } s \overset{r(y)}{\longrightarrow})\)

for some \(w \in \text{dom}(s)\)

This definition states that a synchronisation error can be caused in two ways. The first
way occurs when a formula \( \varphi \) communicates a synchronous adaptation action \( \text{s}(w) \to \) to the system actors defined in \( w \), but one (or more) of these actors are active (i.e., there exists some \( i \in w \) such that \( i : \circ \)), and is thus not in a position to correctly react to the reported adaptation. Similarly, the second way occurs when a formula \( \varphi \) communicates a release action instead, i.e., \( \text{r}(w) \to \).

### 4.6 Conclusion

In this chapter we addressed our second objective (i.e., objective (ii) in Sec. 1.1) by designing and implementing an effective Runtime Adaptation framework for actor systems as an extension of an existing RV setup. In fact we have identified two classes of adaptation actions, namely asynchronous and synchronous adaptations, where the latter require a degree of synchronisation in order to be applied over the respective adaptees, as opposed to the former. The required synchronisation was therefore introduced by building upon the efficient incremental synchronous (Hybrid) monitoring approach explored in Chapter 3. In fact the prototype implementation of our Runtime Adaptation framework was developed as an extension of \texttt{detectEr 2.0} i.e., the extended version of \texttt{detectEr} that we created in Chapter 3. This RA framework for actor systems is now called \texttt{adaptEr}.

Furthermore, to be able to better understand the subtle errors that our RA synchronisation protocol may introduce, we developed a formal operational model. This allowed us to identify situations where our runtime adaptation framework may introduce errors in the monitored system, if synchronisation is not properly defined. This model thus enabled us to identify a notion of synchronisation errors which we formalised as Def. 4.5.3. This therefore enables us to address our third objective by showing that these errors can be detected using static analysis techniques, particularly by developing a type system. In Chapter 5 we therefore investigate ways how to develop a type system for assisting the specifier into writing valid scripts, by rejecting scripts containing invalid synchronisation commands that lead to the identified error.

\footnote{The prototype tool implementation is accessible from \url{https://bitbucket.org/casian/adapter}.}
5. Filtering Out Invalid Adaptation Scripts

As we seen in the previous chapter, by contrast to RV scripts, our runtime adaptation scripts are no longer passive. In fact our RA framework permits the specifier to define blocking necessities and synchronous adaptations that could possibly introduce synchronisation errors. We conjecture that there are various types of errors that erroneously defined scripts may introduce. Such errors include:

1. Applying a synchronous adaptation or a release operation on an actor which was not priorly suspended by a blocking necessity.

2. Blocking an actor earlier than required, therefore disabling subsequent events from occurring; a case in point is shown in Ex. 4.2.2 where we had to explicitly specify the internal communication action $i \rightarrow \! \{\text{inc}, x, y\}$ to be able to block actor $i$ after it forwards the increment request to some other internal component.

3. Adaptations may be applied to external systems interacting with the system which might not be have been instrumented by our RA framework. For instance, we could specify an RA script for our running example system in Fig. 1.1 in which we apply an adaptation to the connected external client mapped to term variable $y$.

A static analysis phase can be employed to detect errors in our RA scripts and reject them at compile time, thereby serving as a tool for assisting the specifier into creating
valid adaptation scripts. As a proof of concept, in this chapter we focus on developing a static type system for ruling out scripts that are erroneous \textit{wrt.} to (1) which we have already formalised as Def. 4.5.3 in Sec. 4.5.2. However, we conjecture that similar static analysis techniques may be employed to detect other errors such as (2) and (3).

This chapter is structured in three sections. In Sec. 5.1 we develop a static type system for ruling out scripts that are erroneous \textit{wrt.} error (1), prior to deployment. Since static typechecking is performed \textit{wrt.} certain assumptions, that might not always be honoured by a system execution, in Sec. 5.2 we augment our runtime monitors with dynamic checks that halt monitoring in case the system deviates from our assumptions. This further prevents synchronisation errors from being introduced by our RA monitors. Finally, in Sec. 5.3 we evaluate our typing strategies by proving \textit{type soundness}, i.e., we prove that at no point during execution can an accepted typed script introduce synchronisation errors in the monitored system.

### 5.1 Static Type Checking

Recall that in Ex. 4.5.4 we presented cases where invalid synchronisation leads to erroneous (stuck) configurations in which synchronous adaptations or release operations are incapable of being validly applied on an active system actor. Particularly, we showed how RA scripts containing concurrent conjunction branches can introduce errors which can be difficult for the specifier to notice prior to deployment, without any form of assistance.

In the previous chapter we formalised these synchronisation errors as Def. 4.5.3 (re-stated below). In this section we therefore focus on developing a static type system that assists the specifier into writing valid scripts by rejecting the scripts which they may violate this formalised notion of synchronisation errors, and are thus \textit{unsafe}.

\textbf{Def. 4.5.3} \textit{error}(s \triangleright \varphi) \overset{\text{def}}{=} (\varphi \overset{s(w)}{\rightarrow} \text{ and } s \overset{s(y)}{\rightarrow}) \text{ or } (\varphi \overset{r(w)}{\rightarrow} \text{ and } s \overset{r(y)}{\rightarrow})

for some \(w \in \text{dom}(s)\)

From this definition note that these synchronisation errors are caused due to (one or
5. Filtering Out Invalid Adaptation Scripts

more) invalid entries specified in the list of actor references, used for applying a syn-
chronous adaptation or release operations. We note that an invalid entry \( i \), defined in a
list of actor references \( w \), can have the following three forms:

- it can either refer to an actor which is currently active, \( \text{i.e.}, \) an actor which is not
  blocked by the instrumented code, \( i:0; \)
- it might not be referring to a process identifier pertaining to the system, \( \text{i.e.}, \) it might
  refer to some generic data, \( i \notin s; \) or else
- it might be referring to some generic data, \( i=3 \notin s. \)

Hence to begin with, our type system must be able to make a distinction between generic
data and actual process identifiers. Moreover, the type system must also be able to
distinguish between two generic classes of process identifiers specified within the RA
script it is analysing.

The first class includes the process identifiers that are used in the synchronous adap-
tations and synchronisation operations (\( \text{i.e.}, \) blocking and releasing) of the RA script
being analysed. As these operations may potentially contribute to synchronisation er-
rors, the type system must ensure that the identifiers pertaining to this class are carefully
used within the script. Hence our type system must employ a mechanism for keeping
track of how these identifiers are being blocked and released at runtime. For instance the
type system must be able to statically track that these identifiers are being blocked by a
blocking necessity that specified \textit{prior} to the application of a synchronous adaptation or
release operation.

More importantly, the type system must ensure that these identifiers are \textit{carefully}
specified within concurrent branches, thereby being able to reject cases where two (or
more) branches exhibit a \textit{race condition}. Race conditions such as those exhibited by \( \varphi'''' \)
in Ex. \[4.5.4\] include for instance cases where a branch may only apply a synchronous
adaptation correctly over an actor \( i \) if it (non-deterministically) executes \textit{before} another
concurrent branch which releases \( i \). Ensuring careful use of these identifiers requires
introducing an element of \textit{linearity}, whereby the use of a process id (pertaining to this
5. Filtering Out Invalid Adaptation Scripts

class) must be restricted to a single monitoring branch. We thus refer to process identifiers pertaining to this class as linear.

The second class includes other process identifiers which are not involved with any kind of operation that requires synchronisation, i.e., these typically used as the subjects of asynchronous necessities and thus never used as subjects of blocking necessities or as part of release lists or adaptation lists. As these process identifiers are not used in such sensitive operations, they can safely be specified amongst concurrent branches (unlike linear process ids). Hence we refer to the process ids pertaining to this class as unrestricted. The type system must however make sure that processes that are assumed to be unrestricted are in fact not being used in synchronisation operations and synchronous adaptations.

We present our static type system in the four subsections: in Sec. 5.1.1 we explain the type annotations introduced by our type system; followed by Sec. 5.1.2 in which we explain the type environments containing the required assumptions for type checking a script. Finally in Sec. 5.1.3 and 5.1.4 we present details explaining the typing rules forming our type system.

5.1.1 The Type Annotations

The type system relies on the type structure defined in Fig. 5.1 (below) where values are partitioned into either generic data, dat, or actor identifiers, where the latter are further subdivided into unrestricted, uid, and linear, lid. The type system is substructural [64], in the sense that it uses linear types to statically track how the linear actor identifiers used for synchronous adaptations, are blocked and released by the parallel branches (i.e., conjunctions) of the respective script.

To be able to statically keep track of how the linear identifiers specified in a given script, are blocked and released, type checking (internally) uses a sub-category for linear types, lbid, to denote a blocked linear identifier. The general idea behind this tracking mechanism is that the type system uses a look-up table to statically approximate how a system actor is specified to blocked and released at runtime; this is achieved by switching
Type Structure

\[ T, U \in \text{Typ} ::= \text{dat} \text{ (generic)} \mid \text{uid} \text{ (unrestricted)} \mid \text{lid} \text{ (linear)} \mid \text{lbid} \text{ (blocked)} \]

Type Environment Splitting

\[
\begin{align*}
\text{SE} & \quad \frac{sE + sE = \emptyset}{\emptyset + \emptyset = \emptyset} \\
\text{SU} & \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \quad T \in \{\text{dat, uid}\}}{(\Gamma_1, l:T) + (\Gamma_2, l:T) = (\Gamma_3, l:T)} \\
\text{SL} & \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \quad T \in \{\text{lid, lbid}\}}{(\Gamma_1, l:T) + \Gamma_2 = (\Gamma_3, l:T)} \\
\text{SR} & \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3 \quad T \in \Gamma_1 + \{\text{lid, lbid}\}}{(\Gamma_2, l:T) = (\Gamma_3, l:T)}
\end{align*}
\]

Adaptation-Script Typing Rules

\[
\begin{align*}
\text{tNCa} & \quad \frac{\Sigma; (\Gamma, \text{bind}(e)) \vdash \varphi, \Sigma; \Gamma \vdash \text{rel}(r) \text{ tt}}{\Sigma; \Gamma \vdash [e]_r \varphi} \\
\text{tFIs} & \quad \frac{\Sigma; \Gamma \vdash \text{ff}}{\Sigma; \Gamma \vdash \text{t}} \\
\text{tTru} & \quad \frac{\Sigma; \Gamma \vdash \varphi}{\Sigma; \Gamma \vdash \psi}
\end{align*}
\]

\[
\begin{align*}
\text{tNCb} & \quad \frac{\text{subj}(e) = l}{\text{sub}(e) = l} \\
\text{tREL} & \quad \frac{\Gamma = \Gamma', w : \text{lid}}{\Sigma; \Gamma \vdash \text{rel}(r) \text{ tt}} \\
\text{tABS} & \quad \frac{\Sigma; \Gamma \vdash \text{rel}(r) \varphi}{\Sigma; \Gamma \vdash \text{aA}(w), \varphi}
\end{align*}
\]

\[
\begin{align*}
\text{tCN1} & \quad \frac{\text{excl}(\varphi, \psi) = \bot}{\Sigma; (\Gamma_1 \vdash \varphi) \Sigma; (\Gamma_2 \vdash \psi)} \\
\text{tCN2} & \quad \frac{\text{excl}(\varphi, \psi) = \langle r_x, r_y \rangle}{\Sigma; \text{eff}(\Gamma, r_x) \vdash \varphi} \\
\text{tMAX} & \quad \frac{(\Sigma, X \mapsto \Gamma); \Gamma \vdash \varphi}{\Sigma; \Gamma \vdash \text{max} X \varphi}
\end{align*}
\]

\[
\begin{align*}
\text{tVAR} & \quad \frac{\Sigma(X) \subseteq \Gamma}{\Sigma; \Gamma \vdash X}
\end{align*}
\]

Figure 5.1: A Type System for Adaptation shML scripts

between types lid and lbid. In fact, whenever the type system analyses an identifier \( i \) that is used in a blocking operation (i.e., a blocking necessity), it consults its look-up table and checks that \( i \) is linear typed, i.e., \( i : \text{lid} \); if this is true, then it changes the type associated with the identifier in its look-up table, from lid to lbid. In this way the type system can detect whether a synchronous adaptation or a release operation is being correctly applied to \( i \), by consulting its look-up table and checking that the associated type is lbid. This signifies that at runtime \( i \) would have been blocked in some prior
5. Filtering Out Invalid Adaptation Scripts

operation as required.

Type checking works on *typed scripts*, where the syntax of Fig. [4.7] is extended so that the binding term variables used in action patterns are annotated by the types `dat`, `uid` or `lid`. Note that the type `lbid` is not part of the external syntax, *i.e.*, this cannot be specified as a type annotation in a script, because this type is *reserved* for internal use by the type system, for tracking purposes.

**Example 5.1.1.** The adaptation-script (4.7) of Ex. 4.2.2 would be annotated as follows:

\[
\varphi' \overset{\text{def}}{=} \max Y. \left[ i?\{\text{inc}, x:\text{dat}, y:\text{uid}\}\right]_a^a \left[ i>_!\{\text{inc}, x, y\}\right]_b^b \left( ([j>_!\{\text{res}, x+1\}]_e^e Y) \land ([z:_!\text{err}]_i^i \operatorname{restr}(i) \operatorname{prg}(z) \{ z \Rightarrow Y \}) \right) \tag{5.1}
\]

In (5.1) above, pattern variables \(x, y\) and \(z\) are associated to types `dat`, `uid` and `lid` respectively.

Note that explicit type annotations are only required to be specified for *binding term variables*, *i.e.*, they are only used within necessities that *first bind* a variable. For example, in (5.1), we do not add the type annotations for variables \(x\) and \(y\) in the second necessity, *i.e.*, \([i>_!\{\text{inc}, x, y\}]_b^b\), as these variables are bound (and were thus annotated) in the prior necessity, *i.e.*, \([i?\{\text{inc}, x:\text{dat}, y:\text{uid}\}]_a^a\).

In addition, note that we do not include explicit type annotations for *statically known* process ids such as \(i\) and \(j\); these typically represent process identifiers that are *registered* and can thus be know at compile time. The necessary type annotations for these registered processes are passed over to the type system as type assumptions represented in the form of a type environment.

### 5.1.2 The Type Environments

Our type system for (typed) adaptation-scripts is defined as the least relation satisfying the rules in Fig. [5.1]. The type system judgements take the form \(\Sigma; \Gamma \vdash \varphi\) where

- Value environments, \(\Gamma \in \mathbf{ENV} :: \pi \mathbf{ID} \cup \mathbf{VAR} \rightarrow \mathbf{TYP}\), map identifiers and variables to types — these are used as a look-up table containing the necessary type assumptions,
5. Filtering Out Invalid Adaptation Scripts

which the system consults and updates, while it compositionally analyses a given typed script.

- Formula environments, \(\Sigma \in \text{LVAR} \rightarrow \text{ENV}\), map formula variables to value environments, e.g., \((\Sigma, \{X \mapsto \Gamma\})\) — these are used to analyse recursive formulas in a compositional way. (The use of this environment will be clarified when we discuss rules \(\tau\text{MAX}\) and \(\tau\text{VAR}\) in Sec. 5.1.3).

Note that we sometimes write \(\Gamma \vdash \varphi\) in lieu of \(\emptyset \vdash \varphi\). We also write \(w : T\) to denote the list of mappings \(l_1 : T, \ldots, l_n : T\) where \(w = l_1, \ldots, l_n\) (recall that metavariable \(l \in (\Pi \cup \text{VAR})\) ranges over identifiers and variables).

**Extending the Type Environments**

The rules in Fig. 5.1 assume standard *environment extensions*, \((\Gamma, \Gamma')\). This allows the type system to add new type assumptions by extending its current value environment with new information. Since value environments are maps, they are therefore implicitly restricted such that a domain value of the map (e.g., \(i\) or \(x\)) can only map to one type, e.g., value environment \(\Gamma = \{i : \text{uid}, j : \text{uid}\}\) is invalid as \(i\) is mapped to two different types. This restriction also reflects in the way a value environment is extended.

For instance, if a valid value environment \(\Gamma = \{i : \text{uid}, j : \text{uid}\}\) is extended by \(\Gamma' = \{i : \text{uid}, z : \text{lid}\}\), we get the resultant environment \((\Gamma, \Gamma') = \{i : \text{uid}, j : \text{uid}, z : \text{lid}\}\). Although identifier \(i\) is present in both \(\Gamma\) and \(\Gamma'\), the resultant extended environment \((\Gamma, \Gamma')\) is still valid as both environments map \(i\) to the same type, i.e., to \(\text{uid}\). Hence the implicit mapping restriction is not breached by the extension. By contrast if \(\Gamma\) is extended by \(\Gamma'' = \{i : \text{lid}, z : \text{lid}\}\), the resultant environment \((\Gamma, \Gamma'')\) would not constitute a valid map.

A similar environment extension is also used for formula environments which map formula variables, e.g., \(X\), onto a value environment e.g., \(\Gamma\).
5. Filtering Out Invalid Adaptation Scripts

Splitting Value Environments

The rules in Fig. 5.1 also use environment splitting, $\Gamma_1 + \Gamma_2$, as defined by the environment splitting rules $sE$, $sU$, $sL$ and $sR$. These rules are used to distribute linearly the mappings amongst two (possibly) smaller environments. Rule $sE$ states that an empty value environment $\Gamma = \emptyset$ can be divided into two empty environments i.e., $\emptyset + \emptyset$. Rule $sU$ dictates that generic data and unrestricted typed references, e.g., $x : \text{uid}$, can be divided amongst the two resultant value environments. For instance, if $x : \text{uid} \in \Gamma$ and $\Gamma$ is split into $\Gamma_1$ and $\Gamma_2$, i.e., $\Gamma = \Gamma_1 + \Gamma_2$, then $x : \text{uid}$ is distributed to both $\Gamma_1$ and $\Gamma_2$, such that $x : \text{uid} \in \Gamma_1$ and $x : \text{uid} \in \Gamma_2$.

This contrasts rules $sL$ and $sR$ which state that linear typed references, e.g., $i : \text{lid}$ or $i : \text{llid}$, can only be given to one of the two resultant environments. For example, if $z : \text{lid} \in \Gamma$ and this is split into $\Gamma_1$ and $\Gamma_2$ then $x : \text{lid}$ can only be distributed to either $\Gamma_1$ (with rule $sL$), i.e., $z : \text{lid} \in \Gamma_1$ but $z : \text{lid} \notin \Gamma_2$, or else to $\Gamma_2$ (with rule $sR$).

This splitting mechanism is essential for the type system to make sure that linear typed references are carefully used amongst the concurrent conjunction branches, by restricting their use to only one branch.

5.1.3 The Typing Rules

The typing rules presented in Fig. 5.1 serve to statically analyse a typed script in a compositional manner. Hence these rules regularly consult and update the type assumptions, stored in the type environments, such that subsequent type judgements are made wrt. the information gained by previous judgements.

For instance in formula (5.1) (stated in Ex. 5.1.1), necessity $[i \triangleright ! \{ \text{inc}, x, y \}]^b_e$ and other subsequent constructs are typechecked wrt. the information gained from analysing prior constructs. In fact, from the prior necessity $[?\{ \text{inc}, x : \text{dat}, y : \text{uid} \}]^a_e$ the type system can deduce that $x$ and $y$ were bound and were given the types $\text{dat}$ and $\text{uid}$ respectively.

Rules $t\text{Tru}$, $t\text{Fls}$ and $t\text{F}$ are quite straightforward, i.e., truth and falsities always
5. Filtering Out Invalid Adaptation Scripts

typecheck, while the analysis of if-statements is approximated by requiring both branches to typecheck. Furthermore, typing rules that update the type environments of subsequent judgements include rules τNcA, for analysing asynchronous necessities, and rules τNcB for blocking necessities.

These rules are very similar: τNcA updates the value environment Γ with the process bindings (i.e., variables annotated with either uid or lid) introduced by the necessity pattern e. This permits for the continuation formula ϕ (e.g., ϕ is the continuation formula of [e]φ) to be typechecked wrt. these new process bindings (i.e., Σ; (Γ, Γ’)+ ϕ, where Γ’ contains the new process bindings).

It also checks that in case of action mismatch, the resultant actor releases, defined the release list r of the respective necessity, also typecheck (i.e., Σ; Γ ⊢ rel(r) tt). Rule τNcB performs similar checks, but the continuation formula typechecking is prefixed by the blocking of the subject of the pattern, (i.e., Σ; (Γ, Γ’)+ blk(l) ϕ).

**Remark 8.** Both τNcA and τNcB utilise the auxiliary function bnd(e) that extracts a map of process bindings from a necessity pattern as shown below:

\[\text{bnd}(i?\{\text{inc}, x:\text{dat}, y:\text{uid}\}) = \{y:\text{uid}\}\]  
(5.2)

\[\text{bnd}(x:\text{lid}+y:\text{uid}!\{\text{res}, z\}) = \{x:\text{lid}, y:\text{uid}\}\]  
(5.3)

From (5.2) note that this function only returns term variable bindings that are only annotated with a process type i.e., lid or uid. The reason for not considering generic data types is that the type system does not track the use of these variables. In fact the main focus of our type system is to detect cases where a synchronous adaptation or a release operation is applied to an invalid actor id, and not about how the data variables are being used and inspected by if-statements.

Furthermore, from (5.3) note that variable z was not included in the resultant map as it does not have a type annotation. As our analyses assumes closed formulas, by the point the type system inspects necessity pattern \(x:\text{lid}+y:\text{uid}!\{\text{res}, z\}\), variable z would have already been bound and added to the respective value environment.

Rule τNcA then uses the map extracted by this function from a pattern e, to typecheck
5. Filtering Out Invalid Adaptation Scripts

the continuation formula, e.g., $\varphi$, wrt. this extended type environment, i.e., $\Sigma; (\Gamma, \text{bnd}(e)) + \varphi$. The same applies for rule $\tau\text{nCb}$, with the difference that the continuation formula is prefixed by a block of the pattern’s subject, i.e., $\Sigma; (\Gamma, \text{bnd}(e)) + \text{blk}(l) \varphi$.

The typing rules $\tau\text{Blk}$ and $\tau\text{Rel}$ for actor blocking (i.e., $\text{blk}(w)$) and releasing (i.e., $\text{rel}(w)$), enable the system to track which actor references are marked as blocked or not. For instance, whenever rule $\tau\text{Blk}$ is used to analyse script $\text{blk}(w) \varphi$ wrt. a value environment $\Gamma$, this rule consults $\Gamma$ and checks that all actor references in $w$ are present in $\Gamma$ and are typed as linear, i.e., $\Gamma = \Gamma''; w: \text{lid}$. If this condition is satisfied, it updates the $\Gamma$ environment by changing the type of the identifiers enlisted in $w$ to $\text{lbid}$, thereby creating a new environment $\Gamma' = \Gamma''; w: \text{lbid}$. Rule $\tau\text{Blk}$ then typechecks the continuation formula $\varphi$ wrt. $\Gamma'$. In this way the typesystem is able to check whether the synchronous adaptations or release operations declared in $\varphi$, are being applied to actor references that were priorly blocked.

In fact, rule $\tau\text{Rel}$ uses a mechanism similar to that of $\tau\text{Blk}$, whereby it checks that all actor references, defined in a release list $w$, have the type $\text{lbid}$ in $\Gamma$. Hence the type system is able to reject cases where release operations are applied to active processes by consulting the value environment and checking whether their type was changed to $\text{lbid}$ by a previously defined blocking operation. If all actor references enlisted in $w$ are typed as linear blocked ($\text{lbid}$) in the $\Gamma$ value environment, then rule $\tau\text{Rel}$ changes their type back to lid, thus creating a new value environment which is used to typecheck the continuation formula.

Typechecking asynchronous adaptations, $\text{aA}(w)_r$, with $\tau\text{AdA}$, requires the adaptees specified in $w$, to be linearly typed, i.e., $w : \text{lid}$, whereas typechecking synchronous adaptations, $\text{sA}(w)_r$, with rule $\tau\text{AdS}$, requires the adaptees to be linearly blocked, i.e., $w : \text{lbid}$. In this way the type system ensures that synchronous adaptations are only applied on actor references which were blocked by a prior blocking operation (i.e., their type was changed from lid to lbid). Both rules consider the respective released actors when typechecking the continuations, (i.e., $\Sigma; \Gamma \vdash \text{rel}(r) \varphi$), thus also checking that the actors defined in the release list $r$ are also applied in a valid way, prior to checking the
5. Filtering Out Invalid Adaptation Scripts

continuation formula \( \varphi \).

We have two rules for typechecking conjunction formulas, \( \varphi \& \psi \), namely rules \( \tau \text{Cn}1 \) and \( \tau \text{Cn}2 \); for now we only consider the former, as the latter is covered in more detail in Sec. 5.1.4. Recall that since conjunction subformulas may be executing in parallel (as defined by the reduction rules \( \text{rCn}1 \), \( \text{rCn}2 \), \( \text{rCn}3 \) and \( \text{rCn}4 \) from Fig. 4.6 and Fig. 4.7), synchronous adaptations and release actions may lead to race conditions as explained in Ex. 4.5.4 (see (4.59) and (4.60)). Rule \( \tau \text{Cn}1 \) eliminates such cases by typechecking each subformula \( \varphi \) and \( \psi \) wrt. a split environment, \( \Gamma_1 + \Gamma_2 \), as is standard in linear type systems [64] (refer to Sec. 5.1.2). This ensures that linear typed actor references (e.g., \( l : \text{lid} \) or \( l : \text{lbid} \)) are only distributed to a single concurrent branch, thereby eliminating cases where multiple branches attempt to release or apply a synchronous adaptation on the same actor references. Note that conjunction formulas are inspected with rule \( \tau \text{Cn}1 \), if the side-condition \( \text{excl}(\varphi, \psi) = \perp \) holds (otherwise \( \tau \text{Cn}2 \) applies); this side-condition is explained in more depth in Sec. 5.1.4.

While analysing recursive definitions, \( \max X \cdot \varphi \), formula environment \( \Sigma \) also plays a crucial role in preserving linearity. In fact this environment permits rule \( \tau \text{Max} \) to save a copy of the value environment \( \Gamma \) at the point where a formula variable \( X \) is bound by \( \max X \cdot \varphi \) (i.e., \( \Gamma \) becomes \( (\Sigma, \{ X \mapsto \Gamma \}) \)). This then enables rule \( \tau \text{Var} \) to compare the saved copy to the contents of the (possibly different) value environment \( \Gamma' \) at the point where \( X \) is used (i.e., when the formula recurs), by checking that \( \Sigma(X) \subseteq \Gamma' \). Note that we check whether the saved copy \( \Sigma(X) \) is a subset of (or equal to) the current value environment \( \Gamma' \), as during the type derivation the value environment might be extended with new bindings while inspecting necessity operations with rules \( \tau \text{NcA} \) and \( \tau \text{NcB} \). Furthermore, this condition together with environment splitting, allow the type system to eliminate cases where a formula that applies synchronous adaptations (or releases), \( \text{recurs} \) along two (or more) concurrent branches which may eventually lead to race conditions.

Hence, checking formula variables is imperative to eliminate cases in which synchronisation errors may appear at a later stage during runtime due to concurrency and recursion. For instance consider formula (5.4) which may appear to be free of potential
5. Filtering Out Invalid Adaptation Scripts

synchronisation errors, yet Ex. 5.1.2 proves otherwise.

$$\varphi_{err} = \max X. [i?3]_e^b ((sA(i),X) \& (X))$$  \hspace{1cm} (5.4)

Example 5.1.2. Assume that when formula $\varphi_{err}$ is used to verify trace $t = i?3; i?3; \ldots$ wrt. $s = (i: \circ)$, their configuration yields the following execution (there may be others).

$$s \triangleright \varphi_{err} \xrightarrow{\tau} (i: \circ) \triangleright [i?3]_e^b ((sA(i),\varphi_{err}) \& \varphi_{err})$$  \hspace{1cm} (5.5)

$$\xrightarrow{\eta_3} (i: \circ) \triangleright \text{blk}(i)((sA(i),\varphi_{err}) \& \varphi_{err})$$  \hspace{1cm} (5.6)

$$\xrightarrow{\tau} (i: \bullet) \triangleright (sA(i),\varphi_{err}) \& \varphi_{err}$$  \hspace{1cm} (5.7)

$$\xrightarrow{\tau} (i: \bullet) \triangleright (\text{rel}(i) \varphi_{err}) \& \varphi_{err}$$  \hspace{1cm} (5.8)

$$\xrightarrow{\tau} (i: \circ) \triangleright ([i?3]_e^b ((sA(i),\varphi_{err}) \& \varphi_{err})) \& ([i?3]_e^b ((sA(i),\varphi_{err}) \& \varphi_{err}))$$  \hspace{1cm} (5.9)

$$\xrightarrow{\eta_3} (i: \bullet) \triangleright ((sA(i),\varphi_{err}) \& \varphi_{err}) \& ((sA(i),\varphi_{err}) \& \varphi_{err})$$  \hspace{1cm} (5.10)

$$\xrightarrow{\tau} (i: \circ) \triangleright (\varphi_{err} \& \varphi_{err}) \& ((sA(i),\varphi_{err}) \& \varphi_{err})$$  \hspace{1cm} (5.11)

The above derivation shows that although formula $\varphi_{err}$ does not immediately yield a stuck configuration, it still manages to get stuck after unfolding formula variable $X$ twice as a result of receiving system event $i?3$ twice in a row. More specifically, in the above derivation notice that after unfolding the recursive definition in (5.5), the resultant formula perceives system action $i?3$ which matches with necessity $[i?3]_e^b$ thus causing the subject actor $i$ to block in (5.6). In (5.7) we see that a synchronous adaptation could validly be applied on actor $i$ as it was already blocked (i.e., $i: \bullet$) in the previous step. This actor is then released in (5.8) as it was specified in the release list of the synchronous adaptation applied in (5.7). In (5.9) we see that after $i$ was released, the recursive definitions of the resultant conjunction formula (i.e., $\varphi_{err} \& \varphi_{err}$) were unfolded. Once again in (5.10), action $i?3$ matches blocking necessities $[i?3]_e^b$ thus causing actor $i$ to block. Finally from (5.10) to (5.11) we see that the synchronous adaptation of the left branch was applied on $i$, and then $i$ was released, i.e., $i: \circ$. Hence the last configuration in (5.11) is stuck since the synchronous adaptation in the right branch is unable to apply on $i$.

This shows how concurrency and recursion can induce synchronisation errors that are very hard to notice without assistance. Therefore, as shown in Derivation [1] our
5. Filtering Out Invalid Adaptation Scripts

typesystem uses the linearity imposed by type environment splitting in \( \tau_{CN1} \), together with the formula environment, \( \Sigma \), to rule out such cases by using rules \( \tau_{MAX} \) and \( \tau_{VAR} \).

**Derivation 1.**

\[
\begin{align*}
\tau_{VAR} & \quad (\Sigma' = \{i : \text{lid}\}) \subseteq (i : \text{lid}) \\
\tau_{REL} & \quad \Sigma'; (i : \text{lid}) \vdash X \quad \text{Rejected!!} \\
\tau_{AdS} & \quad \Sigma'; \Gamma' \vdash (\text{rel}(i) X) \\
\tau_{CN1} & \quad \Sigma'; \Gamma' \vdash (\text{sA}(i), X) \\
\tau_{BLK} & \quad \Sigma'; \Gamma' = \{i : \text{lid}\} \vdash (\text{blk}(i) ((\text{sA}(i), X) \& (X))) \\
\tau_{NCB} & \quad \Sigma'; \Gamma' = \{i : \text{lid}\} \vdash \text{rel}(\varepsilon) \tau \\
\tau_{MAX} & \quad (\Sigma' = \emptyset) \vdash \text{max} X \varepsilon \\
\end{align*}
\]

The above type derivation rejects formula \( \varphi_{err} \) by combining the use of environment splitting with formula environment \( \Sigma \). In particular, when the typesystem analyses the recursive binder \( \text{max} X \) with rule \( \tau_{MAX} \), it saves a copy of \( \Gamma = \{i : \text{lid}\} \), identified by \( X \), in \( \Sigma \). Then while checking conjunction formula \( ((\text{sA}(i), X) \& (X)) \) with rule \( \tau_{CN1} \), the value environment \( \Gamma' = \{i : \text{lid}\} \) is split as \( \Gamma' + \emptyset \). The left branch\(^1\) is thus typechecked wrt. \( \Gamma' \), while the right branch is typechecked wrt. \( \emptyset \). The right branch then gets rejected when rule \( \tau_{VAR} \) is applied to verify that the saved copy *i.e.*, \( \Gamma' = \{i : \text{lid}\} \), is a subset of (or equal to) the assigned value environment *i.e.*, \( \emptyset \); however this is not the case. Hence the formula gets rejected.

\[\blacksquare\]

5.1.4 Introducing Mutual Exclusion

Unfortunately, ensuring that linearity is preserved by using environment splitting (in rule \( \tau_{CN1} \)) is however too coarse of an analysis and rejects useful adaptation-scripts such as

\(^1\)This rule was chosen over \( \tau_{CN2} \) as the side-condition \( \text{exc}((\text{sA}(i), X), (X)) = \perp \) holds; this is discussed in Sec. 5.1.4.

\(^2\)Note that the formula still gets rejected even if the right branch is typechecked wrt. \( \Gamma' \) while the left branch is typechecked wrt. \( \emptyset \).
5. Filtering Out Invalid Adaptation Scripts

(5.1) from Ex. 5.1.1 as shown below.

Example 5.1.3. The conjunction formula used in (5.1) from Ex. 5.1.1 has the form:

\[(\newcommand{\n}[0]{{\Downarrow}}\n\left[ j \n\Downarrow \{\text{res}, x + 1\}\right]_a Y) \& (\left[ z : \text{lid} \n\Downarrow \{\text{err}\}\right]_b \text{restr}(i)_c \text{prg}(z)_d Y)\]

where the subformulas are necessity formulas with mutually exclusive patterns i.e., there is no action satisfying both patterns \(j \n\Downarrow \{\text{res}, x + 1\}\) and \(z : \text{lid} \n\Downarrow \{\text{err}\}\). In such cases, a conjunction formula operates more like an external choice construct rather than a parallel composition \([62]\), where only one branch continues monitoring. Hence these branches can never exhibit a race condition not even due to recursion (as opposed to (5.4) in Ex. 5.1.2). However, if the type system uses rule \(\tau\text{Cn}1\) to analyse this conjunction formula, this gets unnecessarily rejected when rule \(\tau\text{Var}\) is applied to check the formula variable \(Y\) which is used in both branches (i.e., in a similar manner as in Ex. 5.1.2).

To refine our analysis, in Def. 5.1.2 (below), we define an approximating function \(\text{excl}(\varphi, \psi)\) that syntactically analyses subformulas to determine whether they are mutually exclusive or not. When this can be determined statically (as in the case of (5.1)), it means that at runtime at most one branch will continue, whereas the others will terminate due to trivial satisfaction, thereby releasing the actors specified by the respective necessity formulas (recall \(\kappa\text{Nc}3\) from [4.7]). We refer to these releases as side-effects (formally defined in Def. 5.1.1 below).

Accordingly, \(\text{excl}(\varphi, \psi)\) denotes that mutual exclusion can be determined by returning a tuple consisting of two release sets, \(\langle r_\varphi, r_\psi \rangle\), containing the actors released by the respective subformulas when an action is mismatched. Rule \(\tau\text{Cn}2\) then typechecks each subformula wrt. the entire value environment \(\Gamma\), adjusted to take into consideration the side-effects imposed on one branch by the other branch, i.e., it considers actors released by the other (defunct) branch, e.g., \(\Sigma; \text{Eff}(\Gamma, r_\psi) \vdash \varphi\), where \(r_\psi\) signifies the actors that are released in case the other branch \(\psi\) gets trivially satisfied.
5. Filtering Out Invalid Adaptation Scripts

Defining Side-effects

At runtime, whenever a mutually exclusive branch $\psi$ terminates trivially due to mismatching necessities, it releases a number of actors specified in the release-lists of the terminated necessities. These side-effects may therefore effect the matching branch $\varphi$ by for instance releasing a specific linear actor on which this branch applies a synchronous adaptation, i.e., in a similar way to what we have previously shown in Ex. 4.5.4 (see (4.59) and (4.60)).

The $\text{eff}(\Gamma, r)$ function defined in Def. 5.1.1 therefore permits the type system to take into consideration these side-effect releases while typecheck a branch $\varphi$. This means that a branch $\varphi$ is typechecked as if the other exclusive branch $\psi$ has terminated and thus released the respective actors.

Def. 5.1.1 (Side-effects).

\[
\text{eff}(\Gamma, r) = \left\{ l: \text{lid} \mid l \in r \land l: \text{lbid} \in \Gamma \right\} \cup \left\{ l: t \mid l \notin r \land l: t \in \Gamma \right\}
\]

This function therefore takes as input the value environment $\Gamma$ associated with one branch $\varphi$, together with the release set derived from the opposing branch, i.e., $r_\psi$. For each process reference $l$ residing in release set $r_\psi$, this function searches for $l$ in $\Gamma$ and changes its type from $\text{lbid}$ to $\text{lid}$. Each branch (e.g., $\varphi$) can thus be analysed separately with the assumption that the processes defined in the release set of the opposing branch (e.g., $r_\psi$) will actually be released at runtime in case a system event causes this opposing branch (i.e., $\psi$) to terminate trivially.

Defining Mutual Exclusion

The $\text{excl}(\varphi, \psi)$ function, defined in Def. 5.1.2 returns tuple $\langle r_\varphi, r_\psi \rangle$ in case both functions $\text{vexcl}(\varphi, \psi)$ and $\text{vexcl}(\psi, \varphi)$ respectively return release sets $r_\varphi$ and $r_\psi$. These release sets denote the actors released by branches $\varphi$ and $\psi$ respectively, in case these get trivially satisfied at runtime. If either one of these two $\text{vexcl}$ functions return $\bot$, then the $\text{excl}$ function also returns $\bot$, thus denoting that branches $\varphi$ and $\psi$ are not mutually exclusive.
Def. 5.1.2 (Mutual Exclusion).

\[
\text{excl}(\varphi, \psi) = \begin{cases} 
  \langle r_{\varphi}, r_{\psi} \rangle & \text{if } (\text{vexcl}(\varphi, \psi) = r_{\varphi} \neq \bot) \land (\text{vexcl}(\psi, \varphi) = r_{\psi} \neq \bot) \\
  \bot & \text{otherwise}
\end{cases}
\]

\[
\text{vexcl}(\varphi, \psi) = \begin{cases} 
  \text{vexcl}(\varphi_1, \psi) \cup \text{vexcl}(\varphi_2, \psi) & \text{if } \varphi = \varphi_1 \land \varphi_2 \\
  \text{vexcl}(\text{rel}(r) \text{tt}, \psi) & \text{if } \varphi = [e]^{\rho}_{r} \varphi' \land \psi \neq \&_{n=1}^{k} (\text{rel}(r) \text{tt})_n \\
  r & \text{if } \varphi = \text{if }^c \varphi_1 \text{ else } \varphi_2 \land \psi \neq \&_{n=1}^{k} (\text{rel}(r) \text{tt})_n \\
  \text{vexcl}(\varphi', \psi) & \text{if } \varphi = \text{max} \; X. \varphi' \land \psi \neq \&_{n=1}^{k} (\text{rel}(r) \text{tt})_n \\
  r & \text{if } \varphi = \text{rel}(r) \text{ tt} \\
  \epsilon & \text{if } \varphi \neq \text{rel}(r) \text{ tt} \land \psi = \&_{n=1}^{k} (\text{rel}(r) \text{ tt})_n \\
  \bot & \text{otherwise}
\end{cases}
\]

\[
\text{fps}(\varphi) = \begin{cases} 
  e & \text{if } \varphi = [e]^{\rho}_{r} \varphi' \\
  \text{fps}(\varphi_1), \text{fps}(\varphi_2) & \text{if } \varphi \in \{\text{if }^c \varphi \text{ else } \varphi, \varphi \land \psi\} \\
  \text{fps}(\varphi') & \text{if } \varphi = \text{max} \; X. \varphi' \\
  \bot & \text{otherwise}
\end{cases}
\]

The vexcl(\varphi, \psi) function incrementally inspects the contents of a branch \varphi wrt. the opposing branch \psi which is always kept constant during analysis. This function returns the release set of \varphi only if this is mutually exclusive to \psi. We now highlight the three main cases of this function.

The first case we consider is when \varphi is also a conjunction \varphi_1 \land \varphi_2. In this case the vexcl is recursively applied for both \varphi_1 and \varphi_2 wrt. the same opposing branch \psi. The idea is that if \varphi_1 and \varphi_2 are both mutually exclusive to \psi then their conjunction, i.e., \varphi_1 \land \varphi_2 is also mutually exclusive to \psi. In this case the function returns a union of the release sets of both \varphi_1 and \varphi_2, returned by the respective recursive applications. Note however that the inner branches \varphi_1 and \varphi_2 need not be mutually exclusive to each other, as long as they are mutually exclusive to \psi (we explain this in more detail in Ex. 5.1.4).

The second case we consider is when \varphi starts with a necessity, i.e., [e]^{\rho}_{r} \varphi'. In this case the function internally calls the fps(\psi) function to retrieve all the top patterns defined in the opposing branch \psi. To better understand what these top patterns are, as an example assume that we use the fps function to analyse \psi = ([e_1]^{r_1}_{r_1} \psi_1) \land ([e_2]^{r_2}_{r_2} \psi_2) \land (sA(w_3), r_3, \psi_3).
5. Filtering Out Invalid Adaptation Scripts

In this case, \( \text{fps}(\psi) \) returns the pattern set \( \{ e_1, e_2, \_ \} \), where \( e_1 \) and \( e_2 \) refer to the patterns of the guarding necessities of branches \( ([e_1]_{\rho_1}^\psi \_1) \) and \( ([e_2]_{\rho_2}^\psi \_2) \), while \( \_ \) is a universally matching pattern which is returned due to the branch starting with \( \text{sA}(w_3)_3 \) — we sometimes refer to these branches as unguarded.

The \( \text{vexcl} \) function then uses standard pattern matching (as defined by the \( \text{mtch} \) function in Def. 4.5.1) to check whether the pattern \( e \) of the necessity being analysed matches any of the top patterns defined in the opposing branch \( \psi \). If it does not match with any of them (i.e., \( \text{mtch} \) returns \( \bot \) for each top pattern), then this means that branch \( \varphi \) is mutually exclusive to branch \( \psi \). This function is therefore recursively applied for \( \text{rel}(r)_t \) (i.e., the function internally calls \( \text{vexcl}(\text{rel}(r)_t, \psi) \)), thereby denoting that whenever a necessity \( [e]_{\rho}^\psi \) gets trivially satisfied at runtime, it releases the actors enlisted in \( r \) prior to reducing into \( \text{tt} \); this recursive application thus returns \( \text{r} \).

Until now we have ignored the side condition \( \psi \neq \&_{m=1}^k (\text{rel}(r)_n)_n \), which states that for this case to apply, the opposing branch \( \psi \) must not form a collection of trivially satisfied branches e.g., \( \varphi \equiv (\text{rel}(r_1)_t) \& \cdots (\text{rel}(r_n)_t) \). This condition is used throughout most of the cases in the \( \text{vexcl} \) function to navigate its inspection to the penultimate case. This case states that if the formula being inspected, \( \varphi \), is not a release operation and the opposing branch \( \psi \) forms a collection of trivially satisfied branches, i.e., \( \varphi = \&_{m=1}^k (\text{rel}(r)_n)_n \), then the function returns an empty release set, \( \varepsilon \). This condition is thus used to signify that any branch \( \varphi \) is mutually exclusive to a collection of trivially satisfied branches. For instance, if \( \varphi = \text{sA}(w)_s \varphi' \) and \( \psi = \&_{m=1}^k (\text{rel}(r)_n)_n \), then \( \text{vexcl}(\varphi, \psi) \) returns \( \varepsilon \) and not \( \bot \).

Remark 9. These side-conditions therefore allow us to show that two mutually exclusive branches (e.g., \( ([i?3]^b \text{sA}(i), \text{tt}) \& ([i?4]^b \text{sA}(i), \text{ff}) \)), remain mutually exclusive even when one of the branches gets trivially satisfied at runtime. For instance, \( \text{Lemma C.3.4} \) (stated below) proves that if two mutually exclusive branches \( \varphi \) and \( \psi \) reduce over a \( \tau \)-action, then the resultant branches \( \varphi' \) and \( \psi \) remain mutually exclusive to each other.

**Lemma C.3.4** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \varphi \xrightarrow{\tau} \varphi' \) implies \( \text{excl}(\varphi', \psi) = (r_\varphi, r_\psi) \)

This Lemma is proved by rule induction on \( \varphi \xrightarrow{\tau} \varphi' \) in Appendix Sec. C.3, along with two
other similar lemmas, i.e., Lemma C.3.5 and Lemma C.3.6 which respectively prove that mutual exclusion is also preserved in the case of α and µ reductions.

Example 5.1.4. To better understand how the excl function works, consider the following partial script having 4 branches.

\[
\begin{align*}
\text{excl(} & \varphi \mid ([i?3]_j^a \varphi_1 & \mathbin{\&} [x : \text{uid?3}]_k^a \varphi_2) \text{ & } ([i?4]_e^a \psi_1 & \mathbin{\&} [y : \text{uid?5}]_k^a \psi_2) \text{)}
\end{align*}
\]

As shown by the excl function derivation below, if we check for mutual exclusion between the topmost two branches, \( \varphi \) and \( \psi \) we can notice that the patterns in the right topmost branch (i.e., \( i?3 \) and \( x : \text{uid?3} \)) clearly do not match any pattern in the left topmost branch (i.e., \( i?4 \) and \( x : \text{uid?5} \)). Hence this function returns tuple \( ([j;k], l) \) denoting that if \( \varphi \) gets trivially satisfied at runtime, then the actors contained in release set \( (j;k) \) would be released; similarly actor \( l \) is released in case \( \psi \) gets trivially satisfied.

\[
\text{excl(} \varphi \mid ([i?3]_j^a \varphi_1 & \mathbin{\&} [x : \text{uid?3}]_k^a \varphi_2) \text{ & } ([i?4]_e^a \psi_1 & \mathbin{\&} [y : \text{uid?5}]_k^a \psi_2) \text{)}
\]

\[
= \langle \text{vexcl}([i?3]_j^a \varphi_1 & \mathbin{\&} [x : \text{uid?3}]_k^a \varphi_2, \psi) \text{, } \text{vexcl}([i?4]_e^a \psi_1 & \mathbin{\&} [y : \text{uid?5}]_k^a \psi_2, \varphi) \rangle
\]

\[
= \langle \text{vexcl}([i?3]_j^a \varphi_1, \psi) \mathbin{\&} \text{vexcl}([x : \text{uid?3}]_k^a \varphi_2, \psi) \text{, } \text{vexcl}([i?4]_e^a \psi_1, \varphi) \mathbin{\&} \text{vexcl}([y : \text{uid?5}]_k^a \psi_2, \varphi) \rangle
\]

(since \( i?3 \) and \( x : \text{uid?3} \) do not match wrt. \( i?4 \) and \( y : \text{uid?5} \))

\[
= \langle \text{vexcl}(\text{rel}(j) \text{ tt, } \psi) \mathbin{\&} \text{vexcl}(\text{rel}(k) \text{ tt, } \varphi) \text{, } \text{vexcl}(\text{rel}(e) \text{ tt, } \varphi) \mathbin{\&} \text{vexcl}(\text{rel}(k) \text{ tt, } \varphi) \rangle
\]

\[
= \langle \{j \cup \{k \} \} \mathbin{\&} \{e \cup \{l \} \} \rangle
\]

\[
= \langle (j;k) , l \rangle
\]

Therefore with this mechanism we can statically infer that if an event matches either \( i?3 \) or \( x : \text{uid?3} \), it would certainly not match patterns \( i?4 \) and \( x : \text{uid?5} \) (and vice-versa). However, this does not mean that the internal branches are exclusive amongst themselves as well. In fact \( \text{excl}([i?3]_j^a \varphi_1, [x : \text{uid?3}]_k^a \varphi_2) = \bot \) as both patterns may match system action \( i?3 \).

Therefore, as shown in the type derivation (below), while typechecking this partial formula the type system can be less restrictive about the topmost branches \( \varphi \) and \( \psi \) by using type rule \( \tau C n 2 \) (as they are mutually exclusive). However, in the case of inner
Derivation 2.

\[ \begin{array}{c}
\text{tNC} \quad \text{tCN} \quad \text{tCN} \\
\text{tCN1} \quad \text{tCN2} \\
\Sigma; \Gamma_1 \vdash [\ell?3]_t \varphi_1 & \quad \Sigma; \Gamma_2 \vdash [x:\text{uid?3}]_t \varphi_2 \\
\Sigma; \text{eff}(\Gamma, t) \vdash ([\ell?3]_t \varphi_1 \& [x:\text{uid?3}]_t \varphi_2) & \quad \Sigma; \text{eff}(\Gamma, (j; k)) \vdash ([\ell?4]_t \varphi_1 \& [y:\text{uid?5}]_t \varphi_2) \\
\Sigma; \Gamma \vdash ([\ell?3]_t \varphi_1 \& [x:\text{uid?3}]_t \varphi_2) & \quad \Sigma; \Gamma \vdash ([\ell?4]_t \varphi_1 \& [y:\text{uid?5}]_t \varphi_2) \\
\end{array} \]

(where \( \text{eff}(\Gamma, t) = \Gamma_1 + \Gamma_2 \) )

Example Typechecking Derivations

The better understand how our type system accepts valid scripts and rejects invalid ones, we present a number of example type derivations for some of the properties that we have seen so far. For instance in Derivation 3 (below) we show how (5.1) typechecks (i.e., is accepted by the typesystem) wrt. value environment \( \Gamma_0 = \{ i: \text{lid}, j: \text{uid} \} \).

Derivation 3.

\[
\begin{array}{c}
\text{Accetpted} \\
\text{tVAR} \quad \text{tADJ} \quad \text{tREL} \quad \text{tTRU} \\
(\Sigma(Y) = \Gamma_0) \subset \Gamma_7 & \quad \Sigma; \varphi \vdash Y & \quad \Sigma; \Gamma_6 \vdash \varphi & \quad \Sigma; \Gamma_4 \vdash \text{tt} \\
\text{tNC} \quad \text{tBLK} \quad \text{TNCB} \quad \text{tNCA} \\
\Sigma; \Gamma_3 \vdash [j\text{var}](\text{res}, x + 1) \subseteq Y & \quad \Sigma; \Gamma_4 \vdash [z: \text{uid}] \text{restr}(i) \text{prg}(z) \subseteq Y & \quad \Sigma; \Gamma_4 \vdash \text{tt} \subseteq \text{tt} & \quad \emptyset; \max Y. [\ell?\text{inc}, x: \text{dat}, y: \text{uid}] \subseteq (i_0 \vdash [\ell?\text{inc}, x: \text{uid}]_b) \\
\text{tNCB} \quad \text{tBLK} \quad \text{tNCA} \\
\Sigma; \Gamma_1 \vdash \text{blk}(i) [j\text{var}](\text{res}, x + 1) \subseteq Y & \quad \Sigma; \Gamma_3 \vdash [z: \text{uid}] \text{restr}(i) \text{prg}(z) \subseteq Y & \quad \emptyset; \max Y. [\ell?\text{inc}, x: \text{dat}, y: \text{uid}] \subseteq (i_0 \vdash [\ell?\text{inc}, x: \text{uid}]_b) \\
\end{array}
\]

\[\blacksquare\]
5. Filtering Out Invalid Adaptation Scripts

Formula (5.1) is accepted by the type system since all possible three branches in Derivation [3] managed to reach (and satisfy) a base case typing rule (i.e., \( \tau \text{Var} \) and \( \tau \text{Tru} \)).

By contrast, for any value environment \( \Gamma \), the typesystem is unable to typecheck erroneous script \( \varphi'' \) (augmented with the necessary type annotations) from Ex. [4.5.4] and thus rejects it. The derivation below shows how this erroneous script is rejected when typechecked \( \text{wrt. } \Gamma_0 \):

**Derivation 4.**

| Formula Environments: | \( \Sigma = (Y \mapsto \Gamma_0) \) |
|-----------------------|----------------------------------|
| Value Environments:   | \( \Gamma_0 = \{i: \text{lid}, j: \text{uid}\} \) | \( \Gamma_1 = \Gamma_0, \{y: \text{uid}\} \) | \( \Gamma_2 = \text{eff}(\Gamma_1, i) = \Gamma_1 \) | \( \Gamma_3 = \text{eff}(\Gamma_1, e) = \Gamma_1 \) | \( \Gamma_4 = \Gamma_3, \{z: \text{uid}\} \) | \( \Gamma_5 = \{i: \text{lid}, j: \text{uid}, y: \text{uid}, z: \text{lid}\} \) |

\[
\begin{align*}
\tau \text{Var} \quad & (\Sigma(Y) = \Gamma_0) \subset \Gamma_2 \\
\tau \text{NCA} \quad & \Sigma; \Gamma_2 \vdash Y \\
\tau \text{CN2} \quad & \Sigma; \Gamma_2 \vdash [j \mapsto \{\text{res}, x + 1\}]^e_{\Sigma} Y \\
\tau \text{AdS} \quad & \Sigma; \Gamma_3 \vdash \text{restr}(i_y)_{\Sigma} Y \\
\tau \text{BLK} \quad & \Sigma; \Gamma_3 \vdash \text{blk}(z) \text{restr}(i_y)_{\Sigma} Y \\
\tau \text{NCA} \quad & \Sigma; \Gamma_3 \vdash [\{z: \text{uid} \mapsto \text{err}\}]^e_{\Sigma} \text{restr}(i_y)_{\Sigma} Y \\
\tau \text{NCA} \quad & \Sigma; \Gamma_3 \vdash [\{z: \text{uid} \mapsto \text{err}\}]^e_{\Sigma} \text{restr}(i_y)_{\Sigma} Y \\
\tau \text{NCA} \quad & \Sigma; \Gamma_3 \vdash [(j \mapsto \{\text{res}, x + 1\})^e_{\Sigma} Y] \\
\tau \text{MAX} \quad & \emptyset; \Gamma_0 \vdash \max Y. [j \mapsto \{\text{inc}, x: \text{dat}, y: \text{uid}\}]^e_{\Sigma} [\{z: \text{uid} \mapsto \text{err}\}]^e_{\Sigma} \text{restr}(i_y)_{\Sigma} Y
\end{align*}
\]

Rejected!!

\( \text{rel} Y. [j \mapsto \{\text{inc}, x: \text{dat}, y: \text{uid}\}]^e_{\Sigma} [\{z: \text{uid} \mapsto \text{err}\}]^e_{\Sigma} \text{restr}(i_y)_{\Sigma} Y \)

Our type system thus rejects formula \( \varphi'' \) as it detects that actor \( i \) is not being blocked by some blocking necessity prior to applying \( \text{restr}(i_y)_{\Sigma} \). In fact, this error is detected, and thus rejected, by rule \( \tau \text{AdS} \) which detects that the adaptee \( i \) had the type \text{lid} in the value environment \( \Gamma_3 \); if \( i \) had been blocked by a prior construct, then its type would have been changed to \text{lb} and thus the script would have been accepted (as we have previously shown in Derivation [3] for formula (5.1)).

Similarly, the typesystem also rejects scripts that introduce synchronisation errors due to concurrency such as \( \varphi'' \) from Ex. [4.5.4]. Recall that \( \varphi'' \) suffers from a race condition whereby necessity \( [j \mapsto \{\text{res}, x + 1\}]^e_{\Sigma} \) erroneously releases actor \( i \) if it mismatches
a system action. As discussed in Ex. 4.5.4, this results in a premature release of actor \( i \), which may interfere with the synchronous adaptation \( \text{restr}(i)_e \) along the other branch. However, as shown in Derivation 5, rule \( \text{τCn2} \) allows for this interference to be statically detected by considering the side-effects of the respective mutually exclusive branches.

**Derivation 5.**

```
| Formula Environments: \( Σ = (Y \mapsto Γ_0) \) |
|-----------------------------|
| Value Environments: \( Γ_0 = \{i: \text{id}, j: \text{uid}\}; \quad Γ_1 = \{i: \text{uid}, y: \text{uid}\}; \quad Γ_2 = \{i: \text{lid}, j: \text{uid}, y: \text{uid}\}; \) |
| \( Γ_3 = \text{eff}(Γ_2, i) = Γ_1; \quad Γ_4 = \text{eff}(Γ_2, i) = Γ_1; \quad Γ_5 = \{i: \text{lid}\}; \quad Γ_6 = \{i: \text{id}, j: \text{uid}, y: \text{uid}, z: \text{lid}\}. \) |

Rejected!!

\[ i: \text{lid} \notin Γ_6 \]

\[ \text{TVAR} = (Σ(Y) = Γ_0) \subset Γ_3 \]

\[ \text{tBLK} \]

\[ \text{tBLK} \]

\[ \text{tNCB} \]

\[ \text{tNCB} \]

\[ \text{tNCB} \]

\[ \text{tNCB} \]

\[ \text{tMAX} = 0; Γ_0 + \text{MAX Y} \quad \] in fact, when rule \( \text{τCn2} \) is applied in Derivation 5, it uses the excl function to analyse the respective branches. Since these branches are mutually exclusive, the function returns the tuple \( (i, i) \), i.e., the release lists of the top necessities \([j \mapsto y! [\text{res}, x + 1]]_i^a\) and \([z: \text{lid} \mapsto y! \text{err}]_i^b\). The release sets in this tuple are then used to construct environments \( Γ_3 \) and \( Γ_4 \), from \( Γ_2 \), wrt. the side-effects imposed by one branch on the other branch in case of trivial satisfaction. Since both branches release \( i \) upon being trivially satisfied, then both \( Γ_3 \) and \( Γ_4 \) are constructed in the same way, i.e., \( Γ_3 = \text{eff}(Γ_2, i) = \{i: \text{lid}, j: \text{uid}, y: \text{uid}\} = Γ_4 \).

Note that \( i: \text{lid} \) in \( Γ_2 \) was changed to \( i: \text{lid} \) in \( Γ_3 \) and \( Γ_4 \). This allows for the type system to detect that if at runtime the synchronous adaptation \( \text{restr}(i)_e \) is applied, then the trivially satisfied opposing branch (i.e., \([j \mapsto y! [\text{res}, x + 1]]_i^a\)) could have already released actor \( i \), thereby leading to a synchronisation error.
5. Filtering Out Invalid Adaptation Scripts

5.2 Dynamic Analysis of Typed Scripts

The typed adaptation-scripts of Sec. 5.1 need to execute with respect to the systems described in Sec. 4.5. Crucially, however, we cannot expect that a monitored system observes the type discipline assumed by the script. This, in turn, may create type incompatibilities that need to be detected and handled at runtime by the monitor.

Example 5.2.1. Recall that in Derivation 5 the typed script (5.1) (restated below) was accepted by the type system when typechecked with respect to the value environment \( \Gamma_0 = \{ i : \text{lid}, j : \text{uid} \} \).

\[
\varphi' \overset{\text{def}}{=} \max Y. [i?(\text{inc}, x:\text{dat}, y:\text{uid})]_a^a \left( (\exists x_1; y: [\text{res}, x + 1])_a^b Y \right) \land \\
[i^b \cdot \text{inc}, x, y]_b^b \left( \left( \exists z; \text{dat} \cdot y ! \text{err} \lbrace \text{res} \cdot x + 1 \rbrace_1^b \text{prg}(z)_{i, z} Y \right) \right) \quad (5.1)
\]

There are two classes of type incompatibilities that may arise during runtime monitoring:

- When listening for a pattern, e.g., \( i?(\text{inc}, 5, i) \), the system may generate the action \( i?(\text{inc}, 5, i) \); pattern matching the two would incorrectly map the identifier variable \( y \) (of type \( \text{uid} \)) to the linear identifier value \( i \) (which is already typed as \( \text{lid} \)); we call this a type mismatch incompatibility, as an identifier \( i \) can only be assigned one type.

- When listening for pattern \( z:\text{lid} \cdot y ! \text{err} \), the system may generate a matching action \( i^b \cdot \text{err} \) mapping variable \( z \) to \( i \). Aliasing \( z \) with \( i \) violates the linearity assumption associated with \( z \), (i.e., denoted by its type \( \text{lid} \)), which assumes it to be distinct from any other identifier mentioned in the script [64]; we call this an aliasing incompatibility.

Such incompatibilities would not occur if a script is used to monitor a typed system that typechecks with respect to the same value environment used for typechecking the script. For instance, in the case of formula (5.1) which typechecks with respect to \( \Gamma_0 \), the system in Fig. 1.1 would also need to typecheck with respect to the same environment. By employing static typechecking for both the monitor and the system, we would therefore be able to statically detect cases where a system event, such as \( i : \text{lid}?3 \), may erroneously match a necessity pattern such as \( x : \text{uid}?3 \), thereby detecting a type mismatch. A similar technique
can also be used to statically detect that a linear process identifier \( i \), may at runtime be mapped to other linear typed variables in the adaptation formula; thereby detecting cases of aliasing.

In the absence of system typing, our monitors need to perform dynamic type checks (at runtime) and abort monitoring as soon as a type incompatibility is detected. This is important since any violations to the type discipline assumed by the script, may potentially render the adaptations specified in the script unsafe, and thus these should not be administered on the system.

In order to perform dynamic type checks, the monitor should be able to start with an initial set of type assumptions and then extend these assumptions from the information that it learns about the system while handling system events. Hence, the operational semantics of typed scripts is defined \textit{wrt.} the value type environment, \( \Gamma \), with which they are statically typechecked. This environment is then extended at runtime with new type assumptions, that the monitor infers whenever a value received from a system event, is bound to a typed variable defined in a matching necessity pattern. The new type assumption would thus consist in the matched value mapped to the type of the variable it was bound to.

For instance, if a monitor which executes \textit{wrt.} environment \( \Gamma \), receives a system event \( i?3 \) which matches necessity pattern \( x:uid?3 \), it would therefore infer a new type assumption in which it associates identifier \( i \) to type \( uid \) (which is the type of variable \( x \) to which \( i \) was bound). The monitor therefore extends \( \Gamma \) with this new assumption, \textit{i.e.}, \( i:uid \), thereby creating environment \( (\Gamma, i:uid) \).

**Example 5.2.2.** As shown in the derivation below, the execution of the typed script \( \varphi' \) in (5.1) \textit{wrt.} system \( s = (i, j, k, h):o \), would use the type environment \( \Gamma = \{i:lid, j:uid\} \) from Ex. 5.1.1 as a look-up table which its consults in order to detect type incompatibilities. For instance the typed script is able determine that an action such as \( i?\{\text{inc}, 5, i\} \) cannot be pattern matched with \( i?\{\text{inc}, x:\text{dat}, y:uid\} \), as this would lead to a type mismatch when \( y:uid \) is mapped as \( i:uid \) (note the mismatching types, \textit{i.e.}, \( y:uid \mapsto i \) conflicts with the assumption \( i:lid \) which is already present in \( \Gamma \)).
5. Filtering Out Invalid Adaptation Scripts

\[ s \Rightarrow (\Gamma, \varphi') \Rightarrow \langle [\Gamma, y: \text{id}], [i \mapsto ![\text{inc}, x, y]^b] \cdots \rangle \{ x \mapsto 5, y \mapsto i \} \]

(By applying substitution \( \{ x \mapsto 5, y \mapsto i \} \), environment \( \Gamma, y: \text{id} \) becomes \( \{ i: \text{lid}, j: \text{id}, i: \text{id} \} \))

\( \Rightarrow \) Type mismatch!!

The above reduction leads to a stuck configuration caused by the type mismatch of \( i \), as this yields an invalid extension to the value environment \( i.e. \), the resultant environment is no longer a map (as previously explained in Sec. 5.1.2).

Conversely as shown in (5.12) of the below derivation, matching pattern \( i?\{\text{inc}, x: \text{dat}, y: \text{id} \} \) with action \( i?\{\text{inc}, 5, h \} \) would not only constitute a valid match, but also allow monitoring to extend the assumed knowledge of the system from \( \Gamma \) to \( \Gamma' = \{ \Gamma, h: \text{id} \} \), where \( h \) is associated to the type of the matched pattern variable \( y \). The extended environment \( \Gamma' \) would however allow the monitor to detect a type mismatch between pattern \( z: \text{lid} \mapsto h! \text{err} \) and action \( h \mapsto h! \text{err} \) as shown in (5.14).

\[ s \Rightarrow (\Gamma, \varphi') \Rightarrow \langle [\Gamma, h: \text{id}], [i \mapsto ![\text{inc}, 5, h]^b] \rangle \left( \left\{ \begin{array}{l} \{ j \mapsto h!\text{res}(5 + 1)^a \varphi' \} \\ \{ z : \text{lid} \mapsto h! \text{err} \}^b \text{restr}(i) \text{prg}(z)_{\varphi'} \end{array} \right\} \right) \]

(5.12)

\[ \Rightarrow (j, k, h) : o, i : \bullet \Rightarrow \langle [\Gamma, h: \text{id}], \text{restr}(i) \text{prg}(z)_{\varphi'} \rangle \{ z \mapsto h \} \]

(5.14)

(If we apply substitution \( \{ z \mapsto h \} \) we get environment \( [\Gamma, h: \text{id}, h: \text{lid}] \))

\( \Rightarrow \) Type mismatch!!

Importantly, the derivation below shows that the monitor is also enabled to detect an aliasing violation between the same pattern, \( z : \text{lid} \mapsto h! \text{err} \), and action \( i \mapsto h! \text{err} \), since variable \( z \) cannot be mapped to \( i \), as this linear identifier is already present in the respective value environment.

\[ \Rightarrow \langle [\Gamma, h: \text{id}, z: \text{lid}], \text{restr}(i) \text{prg}(z)_{\varphi'} \rangle \{ z \mapsto i \} \]

(If we apply substitution \( \{ z \mapsto i \} \) we get environment \( \Gamma' = \{ i: \text{lid}, j: \text{id}, h: \text{id}, i: \text{id} \} \))

Linear identifier \( i: \text{lid} \in \Gamma \) was remapped to \( z: \text{lid} \) \( \Rightarrow \) Aliasing!!
5. Filtering Out Invalid Adaptation Scripts

It would however allow \( z : \text{lid} \triangleright h!\text{err} \) to be matched to action \( k \triangleright h!\text{err} \), which would (again) extend the current type environment to \((\Gamma', k : \text{lid})\) using the script type association \( z : \text{lid} \).

The Preliminaries for Dynamic Typechecking

Although safe, the mechanism discussed in Ex. 5.2.2 turns out to be rather restrictive for recursive properties (using maximal fixpoints). By this we mean that we end up unnecessarily aborting monitoring for the fear of unsafety. Note that, by alpha-conversion, the variable bindings made by a necessity formula under a recursive formula is different for every unfolding of that formula: e.g., unfolding script (5.1) twice yields

\[
[i?\{\text{inc}, x : \text{dat}, y : \text{uid}\}]^a_e \cdots [z : \text{lid} \triangleright y!\text{err}]^b_i \cdots \left( [i?\{\text{inc}, x' : \text{dat}, y' : \text{uid}\}]^a_e \cdots [z' : \text{lid} \triangleright y!\text{err}]^b_i \cdots \varphi' \right)
\]

where the outer bindings \( x, y, z \) are distinct from the inner bindings \( x', y', z' \). More importantly, however, the scope of these bindings extends until the next fixpoint unfolding, and are not used again beyond that point. For instance, \( x, y, z \) above are not used beyond the respective adaptations of the first unfolding. Thus, one possible method for allowing a finer dynamic analysis for adaptation-scripts, especially relating to linearity and aliasing violations, is to employ a mechanism that keeps track of which linear bindings are still in use. In the example above, this would allow us to bind \( k \) twice — once with \( z \) during the first iteration, and another time with \( z' \). This is safe as we know that by the time the second binding occurs (\( z' \)), the first binding (\( z \)) is not in use anymore, i.e., there is no aliasing.

To implement this mechanism, our formalisation uses three additional components.

1. The operational semantics for adaptation-scripts uses an extra scoping environment, \( \Delta \in \text{Pid} \rightarrow \mathcal{P}(\text{LVar}) \), keeping track of the recursion variables under which an identifier binding is introduced, by associating that identifier to a set of formula variables, \( \kappa \in \mathcal{P}(\text{LVar}) \). Environment \( \Delta \) keeps track of the linear identifiers that
are currently in use only. For instance, if $\Gamma = \{ i : \text{uid} \}, \Delta = \emptyset, \varphi = \max Y. (i?x : \text{lid})^a \{ x?3 \}_x^b \text{sA}(x), Y)$, the script would reduce as follows for event $i?j$:

$$s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{i?j} s \triangleright (\Gamma, j : \text{lid}, \Delta' = \{ j : \{ Y \} \}, [j?3]_x^b \text{sA}(j)j \varphi)$$  \hspace{1cm} (5.15)

Hence as identifier $j$ was bound to $x : \text{lid}$, the monitor extended its type assumptions in $\Gamma$ with the assumption that $i$ is a linear process id i.e., $i : \text{lid}$. It also added an entry in the $\Delta'$ environment showing that linear identifier $j$ is scoped under formula variable $Y$ and is thus currently in use. Therefore, if $i$ is rebound to a linear variable while it is still in use, we can detect aliasing by consulting the scoping environment.

(2) To facilitate updates to environment $\Delta$, the patterns in necessity formulas are decorated by sets of formula variables, denoting their respective recursion scope. For instance, if we apply the decoration process on $\max X. ([e]_r^o \ldots \max Y. ([e']_w^o \text{ff}))$ we get $\max X. ([e_{\{X\}]_r^o \ldots \max Y. ([e'_{\{X,Y\}]_w^o \text{ff}}))$. Decoration can easily be performed using a linear scan of the script before executing it.

(3) The runtime syntax uses an additional construct, $\text{clr}(Y)\varphi$, when unfolding a recursive formula. This is required: to mark the end of an unfolding; and upon execution, to remove all identifier entries in $\Delta$ that have the recursion variable $Y$ declared in their respective set of formula variables, $\kappa$, so as to record that they are not in use anymore. For instance, when using this additional construct in (5.15) we obtain the following derivation:

$$s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{i?j} s \triangleright (\Gamma, j : \text{lid}, \{ j : \{ Y \} \}, [j?3]_x^b \text{sA}(j), \text{clr}(Y)\varphi)$$

$$\xrightarrow{j?3} s \triangleright (\Gamma, j : \text{lid}, \{ j : \{ Y \} \}, \text{clr}(Y)\varphi)$$

$$\xrightarrow{\tau} s \triangleright (\Gamma, j : \text{lid}, \emptyset, \varphi)$$  \hspace{1cm} (5.16)

Note that in (5.16) the $\text{clr}(Y)$ was used to clear the contents in $\Delta'$ which were associated with formula variable $Y$.

Moreover, the $\Delta$ environment is first initialised by associating every statically know lin-
5. Filtering Out Invalid Adaptation Scripts

ear process identifier \(i : \text{lid}\) residing in \(\Gamma\) to \(\kappa = \emptyset\), in the following manner:

\[
\Delta = \{ i : \emptyset | \Gamma(i) = \text{lid} \}
\]

For example, if a script typechecks statically wrt. \(\Gamma = \{ i : \text{lid}, j : \text{uid} \}\), the \(\Delta\) environment is initialised as \(\Delta = \{ i : \emptyset \}\). These linear identifiers are therefore \textit{universally scoped} and thus cannot be cleared.

\textbf{Remark 10.} In this scoping mechanism, the identifiers bound to linear variables from different concurrent branches are added to a conceptually global scoping environment \(\Delta\), in which they are associated to the formula variable \(X\) under which they are currently scoped \textit{i.e.}, in use. The bindings associated to formula variable \(X\) are then cleared from \(\Delta\) whenever \(\text{clr}(X)\) is called. This approach may appear to be problematic in the following situations:

(i) A formula multiply defines the same recursive variable, \textit{e.g.}, \((\text{max} X. \phi_1) \& (\text{max} X. \phi_2)\).
   
   Our clearing mechanism may erroneously clear the bindings used in branch \(\text{max} X. \phi_2\) when branch \(\text{max} X. \phi_1\) recurs (and vice-versa).

(ii) A formula recurs on the \textit{same} recursive variable in two (or more) coexisting concurrent branches, \textit{e.g.}, \(\text{max} X. [?3]_a^{b} ((sA(i),X) \& X)\). In such cases the clearing mechanism could clear the bindings or one recurring branch \textit{e.g.}, \((sA(i),X)\) by using \(\text{clr}(X)\), which could erroneously clear the bindings scoped under the other branch.

We solve issue (i) as we work up to \(\alpha\)-conversion \textit{i.e.}, we rename one of the variables by conducting a linear scan of the script, \textit{e.g.}, the renamed script becomes:

\[(\text{max} X. \phi_1) \& (\text{max} Y. \phi_2[X/Y])\]

In the case of issue (ii), recall that in \(\Delta\) we only track \textit{linear identifiers}, \textit{e.g.}, \(i : \text{lid}\), which means that this issue is problematic whenever a formula that \textit{uses a linear identifier} \(i : \text{lid}\), recurs over the \textit{same} formula variable in \textit{multiple non-mutually exclusive branches}. However from Ex. \[5.1.2\] in Sec. \[5.1.3\] recall that these cases are \textit{statically rejected} by our typing rules. Hence, a \textit{well-typed} script can never introduce such an issue. \hfill \blacksquare
5. Filtering Out Invalid Adaptation Scripts

Figure 5.2: Dynamically Typed Adaptation-Script Rules
5. Filtering Out Invalid Adaptation Scripts

\[
\begin{align*}
\text{iACT} & : (\Gamma, \Delta, \varphi) \xrightarrow{\mu} s \xrightarrow{\alpha} s' \quad (\Gamma, \Delta, \varphi) \xrightarrow{\alpha} (\Gamma', \Delta', \varphi') \\
& \quad s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\alpha} s' \triangleright (\Gamma', \Delta', \varphi') \\
\text{iTRM} & : (\Gamma, \Delta, \varphi) \xrightarrow{\mu} s \xrightarrow{\alpha} s' \quad (\Gamma, \Delta, \varphi) \xrightarrow{\alpha} \\
& \quad s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\alpha} s' \triangleright \text{tt} \\
\text{iADA} & : (\Gamma, \Delta, \varphi) \xrightarrow{\mu} (\Gamma', \Delta', \varphi') \quad s \xrightarrow{\tau} s' \\
& \quad s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s' \triangleright (\Gamma', \Delta', \varphi') \\
\text{iSYS} & : (\Gamma, \Delta, \varphi) \xrightarrow{\tau} (\Gamma', \Delta', \varphi') \quad s \xrightarrow{\tau} s' \\
& \quad s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s' \triangleright (\Gamma, \Delta, \varphi) \\
\text{iMON} & : (\Gamma, \Delta, \varphi) \xrightarrow{\tau} (\Gamma', \Delta', \varphi') \quad s \xrightarrow{\tau} \\
& \quad s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s' \triangleright (\Gamma', \Delta', \varphi')
\end{align*}
\]

Figure 5.3: The Instrumentation Transition Rules (adapted from Fig. 4.7)

The Dynamically Typed Model

Fig. 5.2 describes the transition rules for typed adaptation-scripts, defined over triples \( (\Gamma, \Delta, \varphi) \). Along with the system rules of Fig. 4.7 and the instrumentation rules (adapted to triples \( (\Gamma, \Delta, \varphi) \) from Fig. 4.7) of Fig. 5.3, they form the complete operational semantics. By contrast to Fig. 4.6, \( \text{rMax} \) in Fig. 5.2 unfolds a recursive formula to one prefixed by a clear construct, (i.e., \( \text{clr}(X) \max X. \varphi \)), where rule \( \text{rClr} \) is then used to remove all entries in \( \Delta \) associated with \( X \).

The new version of \( \text{rNc1} \) in Fig. 5.2 implicitly checks for type mismatch incompatibilities by requiring that the environment extension, \( (\Gamma, \Gamma') \), is still a map, i.e., conflicting entries e.g., \( i : \text{uid}, i : \text{lid} \) would violate this condition. It also uses the predicate \( \forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \rightarrow x \sigma \neq y \sigma \), to check that the substitution environment, \( \sigma \), derived from matching a necessity pattern \( e \) to a system event, does not map the same process identifier, \( i \), to two or more distinct linear typed variables such as \( x : \text{lid} \) and \( y : \text{lid} \). For instance, if a system event \( \{ \text{id}, i, i \} \) is matched with necessity pattern \( \{ \text{id}, x : \text{lid}, y : \text{lid} \} \), this predicate detects that \( x \neq y \) but \( x \sigma = y \sigma = i \), and thus detects aliasing. Furthermore, this new version of \( \text{rNc1} \) checks that the new bindings, i.e., \( \text{dom}(\Gamma') \), are distinct from the linear identifiers that are currently in use, i.e., \( \text{dom}(\Delta) \), as these would constitute
5. Filtering Out Invalid Adaptation Scripts

by updating aliasing incompatibilities. If no type incompatibilities are detected, the rule transitions separate branches do not overlap. In fact this rule uses predicate $\text{dom}(\Delta) = (\text{dom}(\Delta') \cap \text{dom}(\Delta''))$ to detect cases where the same identifier is simultaneously mapped to two distinct linear typed variables pertaining to the two concurrent branches $\phi$ and $\psi$ (e.g., identifier $i$ is simultaneously mapped to $x:\text{lid}$ defined in $\phi$, and also to $y:\text{lid}$ in $\psi$).

If any of the conditions for $\text{rNc1}$, $\text{rNc2}$ and $\text{rCn1}$ are not satisfied, the adaptation-script blocks and is terminated in an instrumented setup using rule $\text{rTrm}$ from Fig. 5.3, i.e., it aborts as soon as type incompatibilities are detected.

**An example driven explanation for Detecting Aliasing**

As shown by our newly introduced dynamic checks in rules $\text{rNc1}$, $\text{rNc2}$ and $\text{rCn1}$, aliasing can be introduced in two ways: (i) either by remapping a linear process id while it is already in use, or else (ii) by having two or more branches that simultaneously map the same linear process id to different linear typed variables.

**Example 5.2.3.** For the first case consider the following derivation for formula (5.1) (restated below), wrt. system $s = ((i, j, k, h): \circ), \Gamma = \{i:\text{lid}, j:\text{uid}\}, \Delta = \{i : \emptyset\}$. Note that decorations were added accordingly while deriving (5.1).

$$\varphi' \stackrel{Definition}{=} \max Y. \left[\text{inc}, x: \text{dat}, y: \text{uid}\right]_e \left(\left[\text{inc}, x, y\right]_Y\right)$$

$$s \triangleright (\Gamma, \Delta, \varphi') \quad \text{...} \quad \triangleright (\Gamma, \Delta, \left[\text{inc}, 3, h\right])_Y \quad \left(\left[\text{inc}, 3, h\right]_Y\right)$$

$$\triangleright (i: \bullet, (j, k, h): \circ) \triangleright (\Gamma', \Delta', \left(\text{restr}(i)_e \text{prg}(i)_e \text{clr}(Y)\varphi'\right)$$

where $\Gamma' = \{i: \text{lid}\}$ and $\Delta' = \{i : \emptyset, i : Y\}$

But $\text{dom}(\Delta) \cap \text{dom}(\Gamma') = \{i \neq \emptyset \Rightarrow \text{Aliasing Detected!} \}$
5. Filtering Out Invalid Adaptation Scripts

Aliasing is therefore detected since in the penultimate reduction of the above derivation, linear process id \( i \) was remapped by necessity \([z : \text{lid} \rightarrow y ! \text{err}]^b_i\) to linear typed variable \( z : \text{lid} \), while this was still in use as stated by the entry \( \{i : \emptyset\} \) in \( \Delta \). More specifically, aliasing was detected by the dynamic check \( \text{dom} (\Delta) \cap \text{dom} (\Gamma') = \emptyset \) performed by \( \text{rNc2} \). For the second case, consider the following script:

\[
\phi_{\text{alias}} \triangleq \begin{cases} 
(x : \text{lid} ? 3)_x^b sA(x)_x^{tt} & \text{if } z > 0 \text{ then } sA(x)_x^{tt} \\
(y : \text{lid} ? z : \text{dat})^b_y & \text{if } z > 0 \text{ then } sA(x)_x^{tt}
\end{cases}
\]

Although \( \phi_{\text{alias}} \) observes the linearity imposed by the static typing rules given in Fig. 5.1, aliasing across concurrent branches may however lead to unsafety, particularly in the case a system actor e.g., \( i \), commits an input operation of 3, i.e., upon event \( i ? 3 \). The monitor reduction below shows how the global dynamic check made by rule \( \text{rCn1} \) is used to detect aliasing amongst the two concurrent branches (note that we have added the necessary decorations to \( \phi_{\text{alias}} \) in the reductions).

\[
\text{rCn1} \quad (\text{dom}(\Delta') \cap \text{dom}(\Delta'') = \{i\} \neq \text{dom}(\Delta)) \implies \text{Aliasing}!!
\]

\[
\begin{align*}
\text{error}(s \xrightarrow{w} (\Gamma, \Delta, \varphi)) & \triangleq (\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{s(w)} \text{ and } s \xrightarrow{r(w)}) \text{ or } (\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{r(w)} \text{ and } s \xrightarrow{s(w)}) \\
& \text{for some } w \in \text{dom}(s)
\end{align*}
\]

Based on Def. 5.2.1, we prove Type Soundness wrt. the semantics of typed adaptation.
5. Filtering Out Invalid Adaptation Scripts

scripts.

**Theorem 5.2.1 (Type Soundness).** Whenever $\Gamma \vdash \varphi$ then:

$$s \triangleright (\Gamma, \{i: \emptyset \mid \Gamma(i) = \text{lid}\}, \varphi) \overset{t}{\Rightarrow} s' \triangleright (\Gamma', \Delta', \varphi') \implies \neg \text{error}(s' \triangleright (\Gamma', \Delta', \varphi'))$$

This theorem states that if a dynamically checked monitor $\langle \Gamma, \Delta, \varphi \rangle$ (where $\Delta$ is initialised as $\Delta = \{i: \emptyset \mid \Gamma(i) = \text{lid}\}$), is executed wrt. a system $s$ forming configuration $s \triangleright (\Gamma, \Delta, \varphi)$, then no matter how this configuration evolves at runtime over some trace $t$, it can never reach an unsafe configuration, i.e., a configuration that satisfies the error definition given in Def. 5.2.1.

### 5.3 Proving Type Soundness

So far we have relied on examples to show that our typing mechanisms are able to accept valid scripts while rejecting erroneous ones (either statically or dynamically). However, since there are an infinite number of possible scripts and executions that one can express, we want to provide soundness guarantees. We want to show that a synchronisation error can never be introduced by any arbitrary script $\varphi$ which is: statically accepted by our type system, and which executes wrt. a system $s$ and the respective $\Gamma$ and $\Delta$ environments, i.e., as configuration $s \triangleright (\Gamma, \Delta, \varphi)$. We therefore prove type soundness for dynamically typed scripts to ensure that at no point during execution can an accepted script yield a synchronisation error (as defined by Def. 5.2.1). A few comments are in order prior to discussing the proofs.

#### 5.3.1 Preliminaries

When proving type soundness (and the supporting lemmas) we assume closed formulas. This is required as our type system is compositional, i.e., it decomposes a formula into subparts and analyses them separately. For this reason, certain parts of the formula may be taken out context and become open formulas as shown in the example below:
5. Filtering Out Invalid Adaptation Scripts

Theorem 5.1.1 (Theorem of Type Safety). \( \tau_{\text{TS}} \) is sound for systems with respect to type environments \( \Gamma \). It states that whenever a linear actor identifier is arranged as blocked in \( \Gamma \), then it must indeed be blocked in \( s \) for it to be well-formed.

\[
\begin{align*}
\tau_{\text{TS}} \quad & \quad \text{Open Formula} \quad \vdash \quad \text{Closed Formula} \\
(\Sigma_i : \{X \mapsto \Gamma\}) ; \Gamma & \vdash [e_1]_X^a \, X \, \& \, [e_2]_w^a \, \text{ff} \\
\Sigma ; \Gamma & \vdash \text{max} \, X \, [e_1]_X^a \, X \, \& \, [e_2]_w^a \, \text{ff} \\
\text{Closed Formula} & \quad \text{But at runtime, } X \text{ is substituted by} \\
& \quad \text{max} \, X \, (\lbrack e_1 \rbrack_X^a \, X \, \& \, [e_2]_w^a \, \text{ff}), \text{ thus keeping the} \\
& \quad \text{formula closed.} \\
\end{align*}
\]

(The same argument applies for Data Variables.)

Furthermore, as dynamic typechecking introduces a new runtime construct, \( i.e., \, \text{clr}(X) \), we make the following additions:

- We extend our static typesystem (see Fig. 5.1) with type rule \( \tau_{\text{Clr}} \) that we define as follows:

\[
\begin{align*}
\tau_{\text{Clr}} \quad & \quad \Sigma ; \Gamma \vdash \varphi \\
\Sigma ; \Gamma & \vdash \text{clr}(X) \varphi \\
\end{align*}
\]

- We also extend function \( \text{vexcl} \) (defined in Def. 5.1.2) with an extra case which handles the clear construct in the same way as per the maximal fixed points, \( i.e., \) we modify function \( \text{vexcl} \) as follows:

\[
\begin{align*}
\text{vexcl}(\varphi, \psi) & \equiv \left\{ \begin{array}{ll}
\ldots \\
\text{vexcl}(\varphi', \psi) & \text{if } \varphi' \in \{\text{max} \, X \, \varphi', \text{clr}(X) \varphi'\} \land \psi \neq \&_{n=1}(r)(m) \\
\ldots \\
\end{array} \right.
\end{align*}
\]

Furthermore, our proof of type soundness makes use of a well-formedness judgement for systems \( \text{wrt.} \) type environments \( \Gamma \). It states that when a linear actor identifier is annotated as blocked in \( \Gamma \), then it must indeed be blocked in \( s \) for it to be well-formed.

**Def. 5.3.1** (System Well-formedness). \( \text{wf}(s, \Gamma) \equiv \Gamma(i) = \text{lbid} \impliedby \Gamma(i) = \text{lbid} \Rightarrow s(i) = \bullet \)

Note that Def. 5.3.1 is a technical device used to decompose soundness into the respective individual transitions. However, it does not affect our premise that the system itself may not adhere to the typing discipline of the typing environment. Recall also that linear actors annotated as \( \text{lbid} \) are not allowed in initial environments. Thus, by Def. 5.3.1 any system is well-formed \( \text{wrt.} \) an environment \( \Gamma \) where \( \text{lbid} \notin \text{cod}(\Gamma) \).

### 5.3.2 The Top Level Proof

As is standard, type soundness relies on the following two properties, namely Type Safety (Lemma 5.3.1) and Subject Reduction (Lemma 5.3.2).
5. Filtering Out Invalid Adaptation Scripts

Lemma 5.3.1 (Type Safety).

For all $\Delta \cdot \text{wf}(\Gamma, s), \Sigma; \Gamma \vdash \varphi$ implies $\neg \text{error}(s \triangleright (\Gamma, \Delta, \varphi))$

Type Safety (Lemma 5.3.1) ensures that any typed script cannot immediately be in error wrt. well-formed systems, as defined in Def. 5.2.1. This however does not imply that it will not reach an error after some number of transitions.

For this reason, we also need Subject Reduction (Lemma 5.3.2), which states that a typed script and a well-formed system remain typed and well-formed after a single transition.

Lemma 5.3.2 (Subject Reduction).

$\text{wf}(\Gamma, s)$ and $\emptyset; \Gamma \vdash \varphi$, and $s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\gamma} s' \triangleright (\Gamma', \Delta', \varphi')$ implies $\text{wf}(\Gamma', s')$ and $\Gamma' \vdash \varphi'$

Thus by repeatedly applying Lemma 5.3.2 over a sequence of transitions we know that starting from a typed script and a well-formed system will always lead to an eventual typed script and well-formed system. From this latter fact, we can then apply Lemma 5.3.1 to deduce that the resulting configuration cannot be in error.

To prove Theorem 5.2.1 (Type Soundness). Whenever $\Gamma \vdash \varphi$ then:

$s \triangleright (\Gamma, \{i:0 \mid \Gamma(i) = \text{lid}\}, \varphi) \xrightarrow{t} s' \triangleright (\Gamma', \Delta', \varphi')$ implies $\neg \text{error}(s' \triangleright (\Gamma', \Delta', \varphi'))$

Proof. We prove the following statement from which the theorem follows:

$\begin{aligned}
\text{wf}(\Gamma, s) \quad \text{and} \\
\emptyset; \Gamma \vdash \varphi \quad \text{and} \\
s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{i} s' \triangleright (\Gamma', \Delta', \varphi')
\end{aligned}$

implies $\neg \text{error}(s' \triangleright (\Gamma', \Delta', \varphi'))$

This is proved by numerical induction on the number of transitions in $\xrightarrow{i}$, i.e., $|\xrightarrow{i}| = n$ using Type Safety and Subject Reduction as follows:

Case $n=0$: Since $n = 0$ implies that no transitions were performed, we know

$\begin{aligned}
\text{wf}(\Gamma, s) \quad &\quad (5.17) \\
\Sigma; \Gamma \vdash \varphi \quad &\quad (5.18) \\
s' = s \text{ and } \Gamma' = \Gamma \text{ and } \Delta' = \Delta \text{ and } \varphi' = \varphi
\end{aligned}$

125
5. Filtering Out Invalid Adaptation Scripts

By (5.17), (5.18) and Lemma 5.3.1 (Type Safety) we know that for all $\Delta$
\[ \neg \text{error}(s \triangleright \langle \Gamma, \Delta, \varphi \rangle) \] (5.20)
By (5.19) and (5.20) we can deduce
\[ \neg \text{error}(s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle) \] (5.21)
\[ \therefore \text{Case holds by (5.21).} \]

Case $n=k+1$: Since $n = k + 1$, and the rule premises we know
\[ \text{wf}(\Gamma, s) \] (5.22)
\[ \Sigma; \Gamma \vdash \varphi \] (5.23)
\[ s \triangleright \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \Rightarrow s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle \quad \text{where } | \Rightarrow | = k \] (5.24)
By (5.24) and defn of $t$ we know
\[ s \triangleright \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} s'' \triangleright \langle \Gamma'', \Delta'', \varphi'' \rangle \] (5.25)
\[ s'' \triangleright \langle \Gamma'', \Delta'', \varphi'' \rangle \xrightarrow{\iota} s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle \] (5.26)
By (5.22), (5.23), (5.25) and Lemma 5.3.2 (Subject Reduction) we know
\[ \text{wf}(\Gamma'', s'') \] (5.27)
\[ \Sigma''; \Gamma'' \vdash \varphi'' \] (5.28)
By (5.26), (5.27), (5.28) and IH we know
\[ \neg \text{error}(s' \triangleright \langle \Gamma', \Delta', \varphi' \rangle) \] (5.29)
\[ \therefore \text{Case holds by (5.29).} \]

5.3.3 Proving Type Safety

We now consider proving type safety; here we detail the proof for Lemma 5.3.1.

To Prove Lemma 5.3.1 (Type Safety).
\[ \text{wf}(\Gamma, s) \quad \text{and} \quad \Sigma; \Gamma \vdash \varphi \text{ implies } \neg(\text{error}(s \triangleright \langle \Gamma, \Delta, \varphi \rangle)) \]

Proof. By Rule induction on $\Sigma; \Gamma \vdash \varphi$. 

126
5. Filtering Out Invalid Adaptation Scripts

**Case tFls:** From the rule premises we know that for all $\Delta$

$$\Sigma; \Gamma \vdash ff$$  \hspace{1cm} (5.30)

$$\text{wf}(\Gamma, s)$$  \hspace{1cm} (5.31)

From the monitor reduction rules in Fig. 5.2 we know that $\langle \Gamma, \Delta, ff \rangle \frac{s(y)}{r(y)}$ and also $\langle \Gamma, \Delta, ff \rangle \frac{s(y)}{r(y)}$. Hence, this case holds as by Def. 5.2.1 (Synchronisation Error) we know $\neg(error(s \triangleright (\Gamma, \Delta, ff)))$ as required.

**Note:** The proofs for cases $tTru$, $tVar$, $tIf$, $tAddA$, $tMax$, $tClr$, $tNaC$ and $tNaB$ are analogous to that of case $tFls$.

**Case tCN1:** From the rule premises we know that for all $\Delta$

$$\Sigma; \Gamma \vdash \varphi \& \psi$$  \hspace{1cm} (5.32)

because

$$\Gamma = (\Gamma_1 + \Gamma_2)$$  \hspace{1cm} (5.33)

$$\Sigma; \Gamma_1 \vdash \varphi$$  \hspace{1cm} (5.34)

$$\Sigma; \Gamma_2 \vdash \psi$$  \hspace{1cm} (5.35)

and

$$\text{wf}(\Gamma, s)$$  \hspace{1cm} (5.36)

By (5.36) and system well-formedness (Def. 5.3.1) we know

$$\Gamma(j) = \text{lbid implies } s(j) = \bullet$$  \hspace{1cm} (5.37)

As by (5.33) we know that $\Gamma_1 \subseteq \Gamma$ and $\Gamma_2 \subseteq \Gamma$, then from (5.37) we can deduce

$$\Gamma_1(j) = \text{lbid implies } s(j) = \bullet$$  \hspace{1cm} (5.38)

$$\Gamma_2(j) = \text{lbid implies } s(j) = \bullet$$  \hspace{1cm} (5.39)

By applying the defn of well-formedness (Def. 5.3.1) on both (5.38) and (5.39) we know

$$\text{wf}(\Gamma_1, s)$$  \hspace{1cm} (5.40)

$$\text{wf}(\Gamma_2, s)$$  \hspace{1cm} (5.41)

By (5.34), (5.40) and IH we know

$$\neg(error(s \triangleright (\Gamma_1, \Delta, \varphi)))$$  \hspace{1cm} (5.42)

By (5.35), (5.41) and IH we know

$$\neg(error(s \triangleright (\Gamma_2, \Delta, \psi)))$$  \hspace{1cm} (5.43)
By (5.42) and (5.43) we know that both branches do not yield errors. Hence we can conclude

\[ \neg (\text{error}(s \triangleright (\Gamma, \Delta, \varphi \& \psi))) \]  

(5.44)

\[ \therefore \text{Case holds by (5.44)}. \]

**Case rCN2:** From the rule premises we know that for all \( \Delta \)

\[ \Sigma; \Gamma \vdash \varphi \& \psi \]  

(5.45)

because

\[ \text{excl}(\varphi, \psi) = (r_{\varphi}, r_\psi) \]  

(5.46)

\[ \Sigma; \text{eff}(\Gamma, r_{\varphi}) \vdash \varphi \]  

(5.47)

\[ \Sigma; \text{eff}(\Gamma, r_\psi) \vdash \psi \]  

(5.48)

and

\[ \text{wf}(\Gamma, s) \]  

(5.49)

By (5.49) and system well-formedness (Def. 5.3.1) we know

\[ \Gamma(j) = \text{lbid implies } s(j) = \bullet \]  

(5.50)

By defn of eff we know that eff(\( \Gamma, r \)) contains less processes of type lbid then \( \Gamma \), since eff sets every process reference \( l \in r \) to lid and then produces a new type environment from \( \Gamma \). This allows us to conclude

\[ \forall j: \text{lbid} \in \text{eff}(\Gamma, r_{\varphi}) \Rightarrow j: \text{lbid} \in \Gamma \]  

(5.51)

\[ \forall j: \text{lbid} \in \text{eff}(\Gamma, r_\psi) \Rightarrow j: \text{lbid} \in \Gamma \]  

(5.52)

By (5.50), (5.51), (5.52) and transitivity we can deduce

\[ (\text{eff}(\Gamma, r_{\varphi}))(j) = \text{lbid implies } s(j) = \bullet \]  

(5.53)

\[ (\text{eff}(\Gamma, r_\psi))(j) = \text{lbid implies } s(j) = \bullet \]  

(5.54)

By applying the defn of well-formedness (Def. 5.3.1) on both (5.53) and (5.54) we know

\[ \text{wf}(\text{eff}(\Gamma, r_{\varphi}), s) \]  

(5.55)

\[ \text{wf}(\text{eff}(\Gamma, r_\psi), s) \]  

(5.56)

By (5.47), (5.55) and IH we know

\[ \neg (\text{error}(s \triangleright (\text{eff}(\Gamma, r_{\varphi})), \Delta, \varphi))) \]  

(5.57)

By (5.48), (5.56) and IH we know

\[ \neg (\text{error}(s \triangleright (\text{eff}(\Gamma, r_\psi)), \Delta, \psi))) \]  

(5.58)
By (5.57) and (5.58) we know that both branches do not yield errors. Hence we can conclude

$$\neg(\text{error}(s \triangleright \langle \Gamma, \Delta, \varphi & \psi \rangle))$$

(5.59)

∴ Case holds by (5.59).

**Case tAdS:** From the rule premises we know that for all $\Delta$

$$\Sigma; \Gamma \vdash s\text{A}(w), \varphi$$

(5.60)

because

$$\Gamma = \Gamma', w : \text{lbid}$$

(5.61)

$$\Sigma; \Gamma \vdash \text{rel}(r) \varphi$$

(5.62)

and

$$\text{wf}(\Gamma, s)$$

(5.63)

By (5.63) and *system well-formedness* (Def. 5.3.1) we know

$$\Gamma(f) = \text{lbid implies } s(f) = \bullet$$

(5.64)

From (5.61) and (5.64) we can deduce

$$s(w) = \bullet$$

(5.65)

By (5.61) and rule rAdS we know

$$\langle \Gamma, \Delta, s\text{A}(w), \varphi \rangle \xrightarrow{s(w)} \langle \Gamma, \Delta, \text{rel}(r) \varphi \rangle$$

(5.66)

By (5.65) and rule sAdS we know

$$s \xrightarrow{s(w)} s$$

(5.67)

Hence, by (5.66), (5.67) and Def. 5.2.1 (Synchronisation Error) we know

$$\neg(\text{error}(s \triangleright \langle \Gamma, \Delta, s\text{A}(w), \varphi \rangle))$$

(5.68)

∴ Case holds by (5.68).

**Case tRel:** From the rule premises we know that for all $\Delta$

$$\Sigma; \Gamma \vdash \text{rel}(w) \varphi$$

(5.69)
5. Filtering Out Invalid Adaptation Scripts

because

\[ \Gamma = \Gamma', w : \text{lbid} \]

and

\[ \Sigma; \Gamma', w : \text{lid} + \varphi \]

\[ \text{wf}(\Gamma, s) \]

By (5.72) and system well-formedness (Def. 5.3.1) we know

\[ \Gamma(f) = \text{lbid implies } s(f) = \bullet \]

From (5.70) and (5.73) we can deduce

\[ s(r) = \bullet \]

By (5.70) and rule \textit{rRel} we know

\[ (\Gamma', w : \text{lbid, } \Delta, \text{rel}(w) \varphi) \xrightarrow{\text{r}(w)} (\Gamma', w : \text{lid, } \Delta, \varphi) \]

By (5.74) and rule \textit{sRel} we know

\[ (s, r : \bullet) \xrightarrow{\text{r}(w)} (s, r : \circ) \]

Hence, by (5.75), (5.76) and Def. 5.2.1 (Synchronisation Error) we know

\[ \neg(\text{error}(s \triangleright (\Gamma, \Delta, \text{rel}(w) \varphi))) \]

∴ Case holds by (5.77). \hfill \Box

5.3.4 Proving Subject Reduction

Before proving subject reduction, in Def. 5.3.2 we define the after function – this denotes how a value type environment \( \Gamma \) changes based on the execution of a system action \( \gamma \).

\textbf{Def. 5.3.2 (The after function).}

\[ \text{after}(\Gamma, \gamma) \equiv \begin{cases} 
\Gamma', w : \text{lbid} & \text{if } \gamma = \text{b}(w) \land \Gamma = \Gamma', w : \text{lid} \\
\Gamma', w : \text{lid} & \text{if } \gamma = \text{r}(w) \land \Gamma = \Gamma', w : \text{lbid} \\
\Gamma' & \text{if } \gamma = \alpha \land \exists \Gamma' \cdot \Gamma \subseteq \Gamma' \\
\Gamma & \text{otherwise}
\end{cases} \]

This function enables us to make a more precise deduction of how the runtime reductions, over some arbitrary action \( \gamma \), effect and evolve a value environment \( \Gamma \) into a more
5. Filtering Out Invalid Adaptation Scripts

Updated environment $\Gamma'$ as defined by the function $\text{after}(\Gamma, \gamma)$. This opposes proving that a runtime reduction reduces $\Gamma$ into some arbitrary value environment.

The proof for Subject Reduction relies on the following auxiliary lemmas:

**Lemma C.2.1 (System Reduction).**

$\Gamma' = \text{after}(\Gamma, \gamma)$ and $\text{wf}(\Gamma, s)$ and $s \xrightarrow{\gamma} s'$ implies $\text{wf}(\Gamma', s')$

— This Lemma proves that well-formedness is preserved upon any action $\gamma$. This is proved by rule induction on $s \xrightarrow{\gamma} s'$ in Sec. C.2.

**Lemma C.2.2 (Monitor Reduction).**

$\Sigma; \Gamma \vdash \varphi$ and $\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle$ implies $\Sigma; \Gamma' \vdash \varphi'$ and $\Gamma' = \text{after}(\Gamma, \gamma)$

— This Lemma proves that if a well-typed script performs a dynamically checked runtime reduction, it always reduces into a new well-typed script. This is proved by rule induction on $\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle$; we show two main cases in Sec. 5.3.5 while the rest are proved in Sec. C.2.

**To Prove Lemma 5.3.2 (Subject Reduction).**

$\text{wf}(\Gamma, s)$ and $\emptyset; \Gamma \vdash \varphi$ and $s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\gamma} s' \triangleright (\Gamma', \Delta', \varphi')$ implies $\text{wf}(\Gamma', s')$ and $\emptyset; \Gamma' \vdash \varphi'$.

**Proof.** By Rule induction on $s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\gamma} s' \triangleright (\Gamma', \Delta', \varphi')$.

**Case iAct:** From the rule premises we know

$$s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\alpha} s' \triangleright (\Gamma', \Delta', \varphi')$$  \hspace{1cm} (5.78)

because

$$s \xrightarrow{\alpha} s$$  \hspace{1cm} (5.79)

$$\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle$$  \hspace{1cm} (5.80)

and

$$\text{wf}(\Gamma, s)$$  \hspace{1cm} (5.81)

$$\emptyset; \Gamma \vdash \varphi$$  \hspace{1cm} (5.82)

By (5.80), (5.82) and Lemma C.2.2 (Monitor Reduction) we know

$$\emptyset; \Gamma' \vdash \varphi'$$  \hspace{1cm} (5.83)

$$\Gamma' = \text{after}(\Gamma, \alpha)$$  \hspace{1cm} (5.84)
5. Filtering Out Invalid Adaptation Scripts

By (5.79), (5.81), (5.84) and Lemma C.2.1 (System Reduction) we know
\[ \text{wf}(\Gamma', s) \] (5.85)
∴ Case holds by (5.83) and (5.85).

**Case **SYS: From the rule premises we know

\[ s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s' \triangleright (\Gamma, \Delta, \varphi) \] (5.86)

because
\[ s \xrightarrow{\tau} s' \] (5.87)

and
\[ \text{wf}(\Gamma, s) \] (5.88)
\[ \emptyset; \Gamma \vdash \varphi \] (5.89)

By defn of after (Def. 5.3.2) and since \( \gamma = \tau \) we know that
\[ \text{after}(\Gamma, \tau) = \Gamma \] (5.90)

By (5.87), (5.88), (5.90) and Lemma C.2.1 (System Reduction) we know
\[ \text{wf}(\Gamma, s') \] (5.91)
∴ Case holds by (5.89) and (5.91).

**Case **MON: From the rule premises we know

\[ s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s \triangleright (\Gamma', \Delta', \varphi') \] (5.92)

because
\[ \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\tau} \langle \Gamma', \Delta', \varphi' \rangle \] (5.93)

and
\[ \text{wf}(\Gamma, s) \] (5.94)
\[ \emptyset; \Gamma \vdash \varphi \] (5.95)

By (5.93), (5.95) and Lemma C.2.2 (Monitor Reduction) we know
\[ \emptyset; \Gamma' \vdash \varphi' \] (5.96)
\[ \Gamma' = \text{after}(\Gamma, \tau) \] (5.97)
5. Filtering Out Invalid Adaptation Scripts

By (5.97) and the definition of \( \text{after} \) (Def. 5.3.2) we know
\[
\Gamma' = \Gamma
\]  \hspace{1cm} (5.98)

By (5.94) and system well-formedness (Def. 5.3.1) we know
\[
\Gamma(j) = \text{Id} \text{ implies } s(j) = \bullet
\]  \hspace{1cm} (5.99)

By (5.98) and (5.99) we know
\[
\Gamma'(j) = \text{Id} \text{ implies } s(j) = \bullet
\]  \hspace{1cm} (5.100)

By (5.100) and system well-formedness (Def. 5.3.1) we know
\[
\text{wf}(\Gamma', s)
\]  \hspace{1cm} (5.101)

\( \therefore \) Case holds by (5.96) and (5.101).

**Case ADA:** From the rule premises we know
\[
s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\tau} s' \triangleright (\Gamma', \Delta', \varphi')
\]  \hspace{1cm} (5.102)

because
\[
s \xrightarrow{\mu} s'
\]  \hspace{1cm} (5.103)

\[
(\Gamma, \Delta, \varphi) \xrightarrow{\mu} (\Gamma', \Delta', \varphi')
\]  \hspace{1cm} (5.104)

and
\[
\text{wf}(\Gamma, s)
\]  \hspace{1cm} (5.105)
\[0; \Gamma \vdash \varphi
\]  \hspace{1cm} (5.106)

By (5.104), (5.106) and Lemma C.2.2 (Monitor Reduction) we know
\[
0; \Gamma' \vdash \varphi'
\]  \hspace{1cm} (5.107)
\[
\Gamma' = \text{after}(\Gamma, \mu)
\]  \hspace{1cm} (5.108)

By (5.103), (5.105), (5.108) and Lemma C.2.1 (System Reduction) we know
\[
\text{wf}(\Gamma', s')
\]  \hspace{1cm} (5.109)

\( \therefore \) Case holds by (5.107) and (5.109).

**Case tRM:** From the rule premises we know
\[
s \triangleright (\Gamma, \Delta, \varphi) \xrightarrow{\omega} (\Gamma, \Delta, \tt)
\]  \hspace{1cm} (5.110)
because

\[ s \overset{\alpha}{\rightarrow} s \]  
\[ (\Gamma, \Delta, \varphi) \not\overset{\alpha}{\rightarrow} \]  

(5.111)  
(5.112)

Given that by (5.112) we know that the monitoring stops applying, then we know that the monitor enters an *idempotent state* (i.e., it remains infinitely true) such that it is unable reduce into a script which is *unsound.*

Case holds trivially. □

5.3.5 Proving the Monitor Reduction Lemma

In this section we highlight two main cases pertaining to the proof for Lemma C.2.2 (Monitor Reduction); the other cases are proved in Sec. C.2. The proof for this lemma is conducted by rule induction on the monitor reductions. Particularly, here we show the proofs for cases rNc1 and rMax as these cases rely on the following three auxiliary lemmas:

**Lemma C.2.3** (Formula Substitution).

\[ \Sigma; \Gamma' \vdash \text{max } X \varphi \text{ and } \Sigma; \Gamma \vdash \varphi \implies \Sigma; \Gamma \vdash [\text{clr}(X) \text{max } X \varphi] / X \]

— The Formula Substitution Lemma proves that whenever a formula variable \(X\) is substituted in a well-typed formula, then the resultant formula is also well-typed formula.

**Lemma C.2.4** (Weakening).

\[ (\Sigma; \{X \mapsto \Gamma'\}); \Gamma \vdash \varphi, \; X \notin \text{fv}(\varphi) \implies \Sigma; \Gamma \vdash \varphi \]

— The Weakening lemma proves that if a formula \(\varphi\) which typechecks wrt. formula environment \((\Sigma; \{X \mapsto \Gamma'\})\) has no free occurrences of formula variable \(X\) (i.e., \(X \notin \text{fv}(\varphi)\)), then we can weaken the formula environment by removing the unnecessary entry \(X \mapsto \Gamma'\).

**Lemma C.2.5** (Term Substitution).

\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi \text{ and } \text{mtch}(e, \alpha) = \sigma \text{ and } \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \text{ and } \forall (x, y): \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \implies \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \varphi\sigma \]

— The Term Substitution Lemma proves that whenever a term \(e\) is substituted in a well-typed formula, then the resultant formula is also well-typed formula.
5. Filtering Out Invalid Adaptation Scripts

— The Term Substitution Lemma is used to prove that if a system event $\alpha$ matches a necessity pattern $e$ without introducing any type inconsistencies, then if the substitution environment $\sigma$, created by the match operation, is applied onto a well-typed formula, then the resultant formula is also well-typed.

Note that these three lemmas are proved in Sec. C.2.

**To prove Lemma C.2.2 (Monitor Reduction)**

$\Sigma; \Gamma \vdash \varphi$ and $\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle$ implies $\Sigma; \Gamma' \vdash \varphi' \land \Gamma' = \text{after}(\Gamma, \gamma)$.

*Proof.* By Rule induction on $\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle$.

**Case rMAX:** From our rule premises we know

$$\langle \Gamma, \Delta, \text{max} \ X \varphi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta, \varphi[\text{clr}(X)\text{max} \ X \varphi]/X \rangle$$

$$\Sigma; \Gamma \vdash \text{max} \ X \varphi$$ (5.114)

By (5.114) and rMAX we know

$$(\Sigma; \{X \mapsto \Gamma\}); \Gamma \vdash \varphi$$ (5.115)

By (5.114), (5.115) and Lemma C.2.3 (Formula Substitution) we know

$$(\Sigma; \{X \mapsto \Gamma\}); \Gamma \vdash \varphi[\text{clr}(X)\text{max} \ X \varphi]/X$$ (5.116)

By (5.116), due to substitution we know that every free formula variable $X$ in $\varphi$ is substituted by $(\text{clr}(X)\text{max} \ X \varphi)$, and hence we know

$$X \notin \text{fv}(\varphi[\text{clr}(X)\text{max} \ X \varphi]/X)$$ (5.117)

By (5.116), (5.117) and Lemma C.2.4 (Weakening) we know

$$\Sigma; \Gamma \vdash \varphi[\text{clr}(X)\text{max} \ X \varphi]/X$$ (5.118)

By the defn of after (Def. 5.3.2) and since $\gamma = \tau$ we know

$$\Gamma = \text{after}(\Gamma, \tau)$$ (5.119)

$\therefore$ Case holds by (5.118) and (5.119).

**Case rNc1:** From our rule premises we know

$$\langle \Gamma, \Delta, [e_i]_e^b \varphi \rangle \xrightarrow{\alpha} \langle \Gamma, \text{bnd}(e)\sigma, \Delta', \text{blk}(i) \varphi \sigma \rangle$$ (5.120)
5. Filtering Out Invalid Adaptation Scripts

because

\[ \text{mtch}(e, \alpha) = \sigma \]  
\[ \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \]  
\[ \forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \]  
\[ \text{subj}(\alpha) = i \]

and

\( \Sigma; \Gamma\vdash[e]^b \varphi \) (5.125)

By (5.125) and rNcB we know

\[ \text{subj}(e) = l \] (5.126)
\[ \Sigma; (\Gamma, \text{bnd}(e))\vdash\text{blk}(l) \varphi \] (5.127)
\[ \Sigma; \Gamma\vdash\text{rel}(r) \text{tt} \] (5.128)

By (5.121), (5.122), (5.123), (5.127) and \textit{Lemma C.2.5} (Term Substitution) we know

\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma)\vdash\text{blk}(l\sigma) \varphi\sigma \] (5.129)

By (5.121), (5.124) and (5.126) we know

\[ l\sigma = i \] (5.130)

By (5.129) and (5.130) we know

\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma)\vdash\text{blk}(i) \varphi\sigma \] (5.131)

Since \( \Gamma \subseteq (\Gamma, \text{bnd}(e)\sigma) \), by defn of after (Def. 5.3.2) (Def. 5.3.2) we know

\[ \text{after}(\Gamma, \alpha) = (\Gamma, \text{bnd}(e)\sigma) \] (5.132)

\[ \therefore \text{Case holds by (5.131) and (5.132).} \]

\textbf{Case rNc2:} The proof for this case is analogous to that of case rNc1.

The rest of the cases are proved in Sec. C.2

\[ \Box \]

5.4 Conclusion

In this chapter we have addressed our third objective (i.e., objective (iii) in Sec. 1.1), whereby we developed a static type system for assisting the specifier into writing valid
5. Filtering Out Invalid Adaptation Scripts

runtime adaptation scripts that are free from synchronisation errors (as stated in Def. 4.5.3). This type system distinguishes between identifiers which are used in synchronisation mechanisms and synchronous adaptations i.e., linear identifiers, and those which are not i.e., unrestricted identifiers. By making this distinction, the type system is able to restrict the use of linear identifiers such that errors are not introduce due to race conditions. However it can be less stringent with how the unrestricted identifiers can be used, as these are incapable of introducing synchronisation errors given that they cannot be used in synchronisation operations (if they are the script gets rejected).

The static type system uses a tracking mechanism, in its analysis, to make sure that linear identifiers are always blocked prior to being used in synchronous adaptations. Furthermore, the typesystem detects erroneous scripts that suffer from race conditions by employing a degree of linearity. In this way, unless two (or more) branches are mutually exclusive, the type system allows only one branch to operate on the same linear identifier. This therefore eliminates cases where one concurrent monitoring branch may for instance release a blocked system actor before another branch manages to apply a synchronous adaptation. Implementing this tracking mechanism required augmenting RA scripts with the necessary type annotations, thereby also allowing the type system to distinguish between term variables used for binding generic data, and variables that bind linear and unrestricted process identifiers.

As we monitor untyped systems, process identifiers that are bound to term variables may introduce type inconsistencies such as type mismatches and aliasing. We thus augmented the monitor’s operational LTS semantics with dynamic typechecking so as to eliminate introducing synchronisation errors, due to these type inconsistencies, by aborting monitoring.

Finally we provided a formal proof guaranteeing that our type system is sound wrt. to typed adaptation scripts. We thus guarantee that any typed script that typechecks statically, generates dynamically typed monitors that are safe, i.e., at no point during their execution can they introduce synchronisation errors.
6. Related Work

In this chapter we consider other work available in the current literature which is related to different aspects of this dissertation; we thus structure this chapter in four sections. In Sec. 6.1 we discuss work related to the synchronous and asynchronous monitoring aspects discussed and explored in Chapter 3. Moreover, in relation to the work presented in Chapter 4 in Sec. 6.2 we present other work which employs a form of mitigation mechanisms that relate to our runtime adaptation methodology. In conjunction, in Sec. 6.3 we also discuss work which dealt with extending a formal Logic with a notion of adaptations. Finally, in Sec. 6.4 we present work related to the static and dynamic analysis techniques that we presented in Chapter 5.

6.1 Synchronous and Asynchronous Monitoring

By and large, most widely used online RV tools employ synchronous instrumentation \cite{53,25,19,32,8}. There is also a substantial body of work commonly referred to as asynchronous RV \cite{31,30,6}. However, the latter tools and algorithms assume completed traces, generated by complete program executions and stored in a database or a log file. As explained in Sec. 2.2.1 we term these bodies of work as offline, and their aims are considerably different from the work presented in Chapter 3.

There exist a few tools offering both synchronous and asynchronous monitoring, such as MOP \cite{18,19}, JPAX \cite{45,66} and DB-Rover \cite{34,33}, whereby the speci-
fier can choose whether a property should be monitored for either synchronously or asynchronously. Crucially, however, unlike our newly developed hybrid monitoring approach, these tools do not provide the fine grained facility of supporting both synchronous and asynchronous monitoring at the same time, i.e., by switching between the two modes at runtime. Apart from detectEr [39], eLarva [23] is another Erlang monitoring tool that uses the EVM tracing mechanism to perform asynchronous monitoring.

Rosu et al. [66] make a similar distinction to ours in their definitions synchronous and asynchronous online monitoring, based on which they develop formula rewriting algorithms to implement efficient centralised monitors. However, their definitions of synchrony and asynchrony deals with the timeliness of detections made by their monitors i.e., that the monitors are efficient enough not to lag behind; however, they provide minimal details on how the system is instrumented. By contrast, we focus on showing how instrumentation is carried out over actor-based systems, and show how hybrid instrumentation can be used to obtain timely detections for certain properties, by blocking strategic parts of the monitored system until the monitor performs the necessary checks. This amounts to the definition of synchronous detection monitoring given in [66].

A closer work to ours (wrt. this area) is [26], whereby the authors allow a decoupling between the system and monitor executions but provide explicit mechanisms for pausing the system while the lagging (asynchronous) monitor execution catches up. In our case this mechanism is handled in a more implicit manner as we weave it in a more natural way within our logic. In fact we implement similar mechanisms at a higher level of abstraction by integrating them seamlessly in our logic when switching from asynchronous to synchronous monitoring in our hybrid monitoring approach. Furthermore, in [26] the authors dealt with introducing asynchrony on top of a setting which supported synchronous monitoring, while in our case we had to do the converse. In fact in Chapter[3] we studied ways on how to introduce synchrony in a tool and a setting, which only provide support for asynchrony.

Talcott et al. in [71] compare and contrast three coordination models for actors which cover a wide spectrum of communication mechanisms and coordination strategies. The
6. Related Work

comparison focusses on the level of expressivity of each model, the level of maturity, the level of abstraction of the model, and the way user definable coordination behavior is provided. One of the analysed coordination models, i.e., the Reo model, is a channel based language in which channels may be either synchronous or asynchronous. This model resembles the way our hybrid monitoring protocol interacts with the monitored system. In fact our monitoring language constructs $[\alpha]$ and $[|\alpha|]$, seem analogous to the asynchronous and synchronous channels in the Reo model. The major difference is that these coordination models are used to develop a concurrent system by specifying protocols of communication between its, while our constructs are used for monitoring an existing actor-based system by defining the way that the system and the monitor interact.

6.2 Employing Runtime Mitigation

In [65], Rinard et al. introduce *failure-oblivious-computing*, a tailor made runtime adaptation technique that enables reinforced systems to survive memory errors without memory corruption. The solution consists in the creation of a safe C compiler which inserts bounds checks that *dynamically* detect invalid memory accesses. Upon detection, instrumented systems execute recovery code which mitigates the invalid memory access, thus allowing the system to continue with its normal execution instead of crashing. Similarly, in [59] Rinard et al. present another domain-specific dynamic adaptation technique for *automatic input rectification*. This technique was realized as a prototype SOAP implementation that automatically rectifies potentially dangerous inputs by converting them into typical inputs that the program is more likely to process correctly. Furthermore in [60], Rinard et al. implement yet another dynamic adaptation technique called RCV that enables reinforced systems to survive divide-by-zero and *null dereference* errors. This technique employs a monitor which attaches itself to the target system whenever an error occurs, mitigates its execution, and keeps on tracking the effects of the applied mitigation until they are flushed from the system’s state, in which case the monitor detaches itself.
These techniques can, to some extent, be considered analogous to our adaptation scripts, in the sense that once an adaptation script is compiled, it generates a monitor which dynamically analyses a system and intervenes whenever it detects the specified erroneous behaviour. This is analogous to what the techniques presented by Rinard et al. are doing in order to reinforce their respective systems. The major differences however are that the adaptation properties presented in Rinard et al.’s work target a different domain (i.e., not for actor systems), and they were implemented in a tailor-made fashion, i.e., they were not specified in some high-level logic and then automatically converted into the required adaptation monitor. By contrast, in our work we provide a logic and a tool for specifying runtime adaptation properties for adapting actor-based systems through actor-level manipulations.

Colombo et al. in [27, 22] integrate the notion of compensations within the Larva runtime monitoring framework to allow for the undoing of undesired system actions. Compensations combined with asynchronous monitoring allow for performance overheads to be minimised. This is done by providing the lagging asynchronous monitor with the ability to rollback the system by reverting its state to a point prior to the occurrence of the detected anomaly, even in the case of late detection. This contrasts our adaptations which rather then trying to rollback the system, they acknowledge that an error has occurred and apply mitigation techniques to reduce and possibly eliminate the repercussions of the detected misbehaviour.

Furthermore, compensations are generally user defined and related to a specific action; in fact it is generally assumed that a system action has a compensating action which can be used during the rollback procedure once a property violation is detected. This opposes our work as we define and implement a number of adaptations (i.e., they are not user defined) which are not related to any kind of system action. In fact our adaptations do not assume any information about the code executed by the monitored system. Moreover, Colombo et al.’s work differs in the type of systems that they target. Whereas in our work we target long-running (reactive) actor-based systems, in their work the authors target long-lived transaction systems such as the EntroPay transaction system [27].
6. Related Work

6.3 Extending a Logic with a Notion of Adaptation

In [74] Zhang et al. developed A-LTL, an adaptation-based extension to Linear Temporal Logic (LTL) [20] for formally specifying adaptation requirements of a system in temporal terms. Particularly, the extended logic allows a user to specify the way that an adaptive system can transition from (i.e., adapt) one program (i.e., the source program) into another program (i.e., the target program). This logic was also implemented as a runtime verification tool called AMOeba-RT [42], which provides assurance that dynamically adaptive software satisfies its requirements. AMOeba-RT instruments adaptive programs to insert code which forwards runtime state information to a concurrent monitoring process which verifies whether the adaptive system satisfies the adaptation properties specified in A-LTL and LTL. AMOeba-RT therefore allows for continuous monitoring and verification of post-release behavior of a software system, ensuring that whenever an error occurs the adaptive software either flags an error report, or applies an adaption to mitigate the error automatically.

This opposes our work as our adaptation scripts provide the synthesised monitors with the ability of actually performing the required adaptations, thus converting normal systems into adaptive systems where the monitor takes care of applying the respective adaptations when required. This contrasts the monitors synthesised from A-LTL scripts presented in Zhang et al.’s work, as these are used to verify that an adaptive system applies the necessary adaptations as specified, and thus their monitors (unlike ours) are incapable of performing the adaptations over the monitored system.

6.4 Static Analysis of Monitoring scripts

In [12] Bodden et al. present dependent advices – an AspectJ language extension containing dependency annotations for preserving crucial domain knowledge. This allows dependent advices to execute only when their dependencies are satisfied, thus allowing for optimizations that exploit this knowledge by removing advice-dispatch code from program locations where these dependencies cannot be satisfied. This concept lead to
the creation of CLARA [13] – a static-analysis framework that employs static typestate analyses to automatically convert any AspectJ monitoring aspect into a residual runtime monitor that only monitors events triggered by program locations that the analyses failed to prove safe.

To a certain extent, AspectJ scripts augmented with dependent advices are similar to our typed scripts, as they provide the static compiler with additional information about the environment interacting with the system. In Bodden’s case dependencies dictate whether an aspect will execute at runtime or not, thus allowing for static optimisations, while in our case the types provide information which allows us to statically determine whether a script is well formed or not i.e., whether it introduces synchronisation errors or not. Therefore, our aims differ from those of Bodden et al.’s work, as in our work we aimed to employ static analysis for our RA scripts so as to ensure a degree of correctness. Conversely, in Bodden et al.’s work the main aim was to apply static analysis in order to reduce the overheads imposed by the AspectJ scripts. Furthermore, our work also differs in the way we evaluate our static analysis technique, since we provide type soundness proof as opposed to Bodden et al. which provide empirical performance tests.

Castellani et al. [16] extend previous work by Dezani et al. [72], whereby they present a model (calculus) of self-adaptive, multiparty communications, in which security violations occur when processes attempt to communicate messages using inappropriate security levels. The model consists in three components: global types, monitors, and processes. Global types represent how processes interact and define reading permissions for each process. Monitors are then obtained by projecting a global type onto a process thus obtaining a monitored process; hence global types are in some sense analogous to our typed scripts. Monitors are able to mitigate security violations using local and global adaptations, for handling minor and serious violations, respectively.

Similar to our work, the authors also establish type soundness by which they ensure that global protocols still execute correctly, even when the system adapts itself to preserve security. However, they assume that the processes requiring monitoring are also well-typed, whereas we do not rely on such an assumption. Specifically we consider
the system as a black-box and assume minimal information. This is reflected in the way we model the system in our RA operational model, and in the type assumptions used by our type system, whereby the only information that we assume about the monitored system is whether its actors are active or suspended by our instrumentation. Hence our adaptations can apply over any (instrumented) actor regardless of the code it executes.
7. Conclusion

In this thesis we sought to develop a Runtime Adaptation framework for actor systems by extending an existing Runtime Verification tool, with adaptation functionality that can be localised to individual concurrent system components, thereby allowing for specific actors to be mitigated while leaving the other actors unaffected. This demanded addressing the following challenges:

(i) Effective runtime adaptation on actor systems requires introducing a degree of synchrony in our monitors. As synchrony is not natively supported by the actor model, this requires studying ways for introducing synchronous monitoring in an efficient manner.

(ii) An effective RA framework must also provide a number of adaptations that can be used to mitigate erroneous system behaviour by modifying several architectural aspects of actor systems. We must therefore identify these adaptations and develop a mechanism by which these can be made applicable to any actor system, i.e., regardless of the code that it executes. As RA properties are no longer passive, these may introduce errors at runtime. Formalising the runtime behaviour of our RA scripts thus relieves us from having to deal with the complexities of the implementation. This therefore enables us to identify and better understand the subtle errors that our RA monitors may introduce at runtime.

(iii) Formalisation also enables us to study ways how erroneous RA scripts can be de-
7. Conclusion

tected and rejected prior to being deployed, by employing static analysis techniques. In this way we should therefore provide the specifier with a tool that serves as guidance for writing error-free runtime adaptation scripts. This tool must however provide an element of guarantee about its own correctness.

To address our first objective (i.e., (i)), in Chapter 3 we have therefore studied different techniques for introducing synchronous monitoring on top of an inherently asynchronous platform (i.e., the actor model). After integrating these techniques in detectEr, we carried out a systematic assessment of the relative overheads incurred by these different monitoring techniques. This assessment allowed us to conclude that although an incremental synchronisation approach still introduces a degree of synchronisation overheads, it however permits for these overheads to be minimised by employing synchronisation on a by-need basis.

In light of the obtained performance results, in Chapter 4 we addressed our second objective, (i.e., (ii)). We thus identified a number of adaptation actions that allow for violations to be mitigated by modifying different aspects of actor systems, such as in the case of process restarts and message interceptions. We then categorised these adaptations into two classes, namely asynchronous and synchronous adaptations, where the latter require synchronisation when applied over the respective adaptees. Synchronisation was therefore introduced by building upon the efficient incremental synchronisation methodology identified in the previous chapter. Hence, by adding minimally intrusive annotations, we augmented detectEr’s specification language with synchronisation mechanisms, for blocking and releasing actors, along with implementations of the identified adaptation actions. This allows the specifier to develop adaptation scripts that yield monitors which are capable of mitigating erroneous behaviour by executing adaptations. A prototype implementation of our Runtime Adaptation framework called adaptEr was thus developed as a result of this extension to detectEr.

Furthermore, we also gave a formal specification of the augmented logic (with adapt-

---

1A prototype implementation of this tool can be accessed and downloaded from https://bitbucket.org/casian/adapter.
7. Conclusion

tations) through a formal operational model. This allows us to predict how an adaptation script operates at runtime without actually executing it, and thus enabled us to better understand the subtle errors that our RA monitors may introduce. Hence this model permitted us to identify and formalise situations where our runtime adaptation framework may introduce errors if synchronisation is not properly defined.

Based on this formal model, in Chapter 5 we addressed our third objective (i.e., (iii)) by developing a static type system for assisting the specifier into writing valid runtime adaptation scripts. We identified a number of errors that erroneous RA scripts may introduce at runtime in the monitored system. We then focused on showing how one of these errors i.e., the synchronisation error formalised in Chapter 4, can be statically detected. We finally evaluated our type system by proving type soundness wrt. typed adaptation scripts, thereby guaranteeing that the monitors that are synthesised from a valid adaptation script (i.e., a script which was accepted by our type system), can at no point during execution introduce synchronisation errors.

Once we addressed these three objectives we established that there exists an important interplay between synchronisation and effective adaptation actions. Although this introduces additional overheads, as well as, the possibility of introducing runtime errors, we developed and evaluated concepts which enable adaptations to be performed efficiently, and which provide assistance for writing error-free scripts. Despite implementing these concepts wrt. actor systems developed in Erlang, we conjecture that similar challenges would arise when developing a similar RA framework for other actor-based technologies such as Scala [43] and Akka [1]. Furthermore, these concepts may also be applicable to other inherently asynchronous architectures such as service-oriented systems [48] and distributed systems [29]. Therefore despite requiring certain parts of the implementation to be reformulated for the respective technologies, the core concepts studied in this thesis (e.g., incremental synchronisation, the identified adaptations, etc.) would still be relevant and applicable.
7. Conclusion

7.1 Future Work

In order to further improve our work, we propose the exploration of the following suggestions.

7.1.1 Implementing More Adaptations

In Chapter 4 we have identified a number of adaptation categories which modify different aspects of an actor system. We have however implemented only a representative adaptation action for each category, as a proof-of-concept. We therefore propose that in future work we implement other adaptation actions thus providing the user with more adaptations that can be used for mitigating the detected violations.

7.1.2 An Algorithmic Type System

When the type system developed in Chapter 5 analyses two non-mutually exclusive branches, it preserves linearity by using an environment splitting operation to assign linear identifiers to only one concurrent branch, thereby eliminating cases which could lead to race conditions. Although our type system is implementable, the inherent nondeterminism of environment splitting, does not provide a directly implementable algorithm for splitting the type environment. We conjecture that the development of an Algorithmic Type System [64] in which linear environment splitting can be performed deterministically, should require minimal alterations to our existing type system. Furthermore, to prove that this algorithmic version is also sound, it suffices proving that it corresponds to our existing type system, for which we have already proven type soundness.

7.1.3 Type inference

Although we provide a tool which assists the specifier into writing valid scripts, this requires the user to manually include additional annotations and to provide an environment
containing the initial type assumptions which are used by the type system. We conjecture that the user should ideally be relieved from this additional effort. We therefore propose the exploration of type inference techniques that could be used to automatically infer the required type assumptions and include the respective annotations. Hence a type inference system could be used to analyse an untyped script in order to generate a type environment and a typed script which typechecks wrt. our type system.

7.1.4 Static Detection of Other Errors

In this thesis we aimed to prove the concept that although erroneous RA scripts introduce runtime errors, we can still provide mechanisms for detecting these errors prior to deployment. Although in Chapter 5 we identify a number of errors that our RA monitors may introduce, in order to prove this concept we developed a type system which detects one of these errors i.e., synchronisation errors which we formalised in Chapter 4. However we conjecture that in most cases, addressing the other errors would only require minor alterations to our type analysis techniques.
A. Monitor Optimisations in detectEr

In this chapter we present additional information with regards the different monitor synthesis techniques that are present in the current implementation of the detectEr RV tool. In Sec. A.1 we look into the original monitor synthesis algorithm, presented in [39] and explain its major causes of inefficiencies. Following this in Sec. A.2 we give a brief overview on how static and dynamic optimisations where proposed in [15] and augmented in detectEr as means for lowering these overheads to a more feasible standard.

A.1 Synthesising Unoptimised Monitors

The original synthesis algorithm [39] aims to be modular by generating independently executing monitor combinators for each logical construct, interacting with one another through message passing.\footnote{Every combinator is implemented as a (lightweight) Erlang process (actor) [17], uniquely identifiable by its process ID. Messages sent to a process are received in its dedicated mailbox, and may be read selectively using (standard) pattern-matching.} For instance, the synthesis parallelises the analysis of the subformulas $\varphi_1$ and $\varphi_2$ in a conjunction $\varphi_1 \& \varphi_2$ by:

(i) synthesising concurrent monitor systems for $\varphi_1$ and $\varphi_2$ respectively; and

(ii) creating a conjunction monitor combinator that receives trace events and forks (forwards) them to the independently-executing monitors of $\varphi_1$ and $\varphi_2$.

Since the submonitor systems for $\varphi_1$ and $\varphi_2$ may be arbitrarily complex (needing to analyse a stream of events before reaching a verdict), the conjunction monitor is permanent.
A. Monitor Optimisations in detectEr

in the monitor organisation generated, so as to fork and forward event streams of arbitrary length. It is also worth noting that the synthesis algorithm assumes formulas to be guarded, where recursive variables appear under necessity formulas; this is required to generate monitors that implement lazy unrolling of recursive formulas, thereby minimising overheads.

\[ m = \text{srv} ? x, y \]
\[ m1 = \text{srv} ! z \]
\[ m2 = \text{srv} ! z \]

Figure A.1: Monitor Combinator Generation for Formula (A.1) and its execution wrt. trace srv?{v, c}; srv?{v′, c′}

Example A.1.1. Consider a simple Erlang system implementing a predecessor server receiving messages of the form \((n, \text{clientID})\) and returning \(n - 1\) back to \(\text{clientID}\) whenever \(n > 0\), but reporting the offending client to an error handler actor, \(\text{err}\), whenever \(n = 0\). It may also announce termination of service by sending a message to \(\text{end}\). A
A. Monitor Optimisations in detectEr

safety correctness property in our logic would be:

\[
\max X \left[ \begin{array}{l}
\text{[srv?}(x,y)\text{]} \land \text{([srv-end!}\_\text{ff}) \land \text{([srv-err!}z\text{][if } x \neq 0 \lor y \neq z \text{ then ff else } X))
\end{array} \right]
\]

(A.1)

It is a recursive formula, \( \max X \ldots \), stating that, whenever the server implementation receives a request (input action), \([\text{srv?}(x,y)\ldots \), with value \( x \) and return (client) address \( y \), then it should not:

1. terminate the service (before handing the client request), \([\text{srv-end!}\_\text{ff})\).
2. report an error, \([\text{srv-err!}z\ldots \), when \( x \) is not 0, or with a client other than the offending one, \( y \neq z \).
3. service the client request, \([\text{srv-y!}z\ldots \), with a value other than \( x - 1 \).

These conditions are invariant; maximal fixpoints capture this invariance for server implementations that may never terminate as they are considered correct as long as the conditions above are not violated.

From formula (A.1), the monitor organisation \( m \) (depicted in Fig. A.1) is generated, consisting of one process acting as the combinator for the necessity formula \([\text{srv?}(x,y)]\varphi\).

If an event of the form \( \text{srv?}(v,c) \) is received, the process pattern matches it with \( \text{srv?}(x,y) \) (mapping variables \( x \) and \( y \) to the values \( v \) and \( c \) resp.) and spawns the (dashed) monitor system shown underneath it in Fig. A.1 instantiating the variables \( x, y \) with \( v, c \) resp. The subsystem consisting of three monitor subsystems, one for each subformula guarded by \( \text{srv?}(x,y) \) in (A.1), connected by two conjunction forking monitors. When the next trace event is received, e.g., \( \text{srv=c!}v-1 \) (a server reply to client \( c \) with value \( v-1 \)), the conjunction monitors replicate and forward this event to the three monitor subtrees. Two of these subtrees do not pattern match this event and terminate, while the third subtree (i.e., submonitor \( m2 \)) pattern-matches it and instantiates \( z \) for \( (v-1) \), thereby also evaluating the conditional and unfolding the recursive variable \( X \) to monitor \( m \). If another server request event is received, \( \text{srv?}(v',c') \) (with potentially different client and value arguments \( c' \) and \( v' \)), the conjunction monitors forward it to \( m \), pattern matching it and generating a subsystem with two further conjunction combinators as before.

\[\blacksquare\]
A. Monitor Optimisations in detectEr

Formula \( (A.1) \) is a pathological example, highlighting two inefficiencies of the unoptimised synthesised monitors. First, conjunction monitors mirror closely their syntactic counterpart and can only handle forwarding to two sub-monitors. As a result, formulas with nested conjunctions (as in formula \( (A.1) \)) translate into organisations of cascading conjunction monitors that are inefficient at forwarding trace events.

Second, the current monitor implementation does not perform any monitor reorganisations at runtime. When a conjunction formula \( \varphi_1 \& \varphi_2 \) is translated, the conjunction combinator monitor organisation is kept permanent throughout its execution because it is assumed that the respective sub-monitors for \( \varphi_1 \) and \( \varphi_2 \) are long-lived. This heuristic however does not apply in the case of formula \( (A.1) \), where two out of the three sub-monitors terminate after receiving a single event. This feature, in conjunction with recursive unfolding, creates chains of indirections through conjunction monitors with only one child, as shown in Fig. \( A.1 \) (bottom row).

A.2 Monitor Optimisations

The first optimisation introduced in [15] is that of conjunction monitor combinators that fork-out to an arbitrary number of monitor subsystems. For instance, the corresponding monitor formula \( (A.1) \) would translate into a monitor organisation consisting of one conjunction combinator with three children (instead of two combinators with two children each) as shown in Fig. \( A.2 \) (left). This is more efficient from the point of view of processes created, but also in terms of the number of replicated messages required to perform the necessary event forwarding to monitor subsystems – e.g., the conjunction combinator of Fig. \( A.2 \) generates three message replications to forward an event to the three sub-monitors, as opposed to the unoptimised approach which requires sending four messages (see Fig. \( A.1 \)). This static optimisation is conducted through syntactic manipulation using a pre-processing phase.

The second optimisation [15] builds on the first one by allowing conjunction monitor combinators to dynamically reorganise the monitor configuration so as to keep the event
A. Monitor Optimisations in detectEr

flow structure as efficient as possible. In order to keep overheads low, this reconfiguration operation is kept local, where unaffected monitor subsystems should continue with their runtime analysis while the restructuring is in process. Stated otherwise, the monitor reorganisation happens while trace events are still being received, and the operation needs to guarantee that (i) no trace events are lost (ii) trace events are not reordered. Reorganisations are carried out by conjunction combinators, which are provided with the capability to add and delete monitor subsystems from their internal list of children. For instance, when an event causes a child sub-monitor to terminate, the parent (conjunction) monitor is sent a termination message which allows it to remove the terminated sub-monitor from its child-list.

The dynamic restructuring protocol employed by this optimisation yields monitor organisations with only one (eventual) conjunction node at the root, and a list of monitor subsystems processing the forwarded events (a spider-like configuration). For instance, for the event tracesrv?{v, c}; srv>c!(v − 1); srv?v′, c′}, the synthesised monitor for formula (A.1) yields the evolution shown in Fig. A.2.
B. Additional Properties and Tabulated Results

In this Chapter we present additional information related to the impact assessment experiments conducted in Chapter 3. In Sec. B.1 we provide an overview of two additional properties written for the Yaws webserver, that we used along with properties (3.3) and (3.4) for obtaining the performance results presented in the graphs in Figures 3.6, 3.7 and 3.8 given in Sec. 3.3.3. To provide an even clearer insight of the results presented in these graphs, in Sec. B.2 we represent the same results as tables.

B.1 Additional Runtime Verification Properties

Here we present two other properties developed for the Yaws webserver and used in our experiments while conducting our impact assessment analysis. Here we only present the versions used for asynchronous and synchronous monitoring as the versions for Hybrid monitoring can easily be inferred by converting all the falsities (ff) in the scripts to sff.

B.1.1 Five or more Headers Required Property

When sending a request to a webserver, HTTP clients also send a number of HTTP headers containing information about themselves and about their request. Typically, an
HTTP request should provide about 5 or 6 information headers (http_header), so that the webserver may correctly handle the request and send the appropriate response. A property may be used to check and ensure that any request sent to the server contains at least 5 headers. This behaviour may be verified using either property (B.1) or (B.2).

$$\text{max } X. \left( \begin{array}{l}
\text{[AcceptorPid}} (^！hPid, \text{next, } \_ )] \\
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_req, GET, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

$$\text{& } \left( \begin{array}{l}
hPid \triangleright \text{return} \left( \text{[yaws, do_recv, } 3, \text{[ok, [http_header, \_ ]]} \right) \right)
\end{array} \right)$$

Although these two properties may be considered equivalent, they do not translate
into the same concurrent monitor. In fact the monitor generated from property (B.1) is specified to immediately spawn 6 concurrent submonitors for each connected client, upon receiving an HTTP request notification (http_req) event. Five of these submonitors flag a violation when 4 or less headers are received, otherwise they terminate. The sixth monitor reapplies the property for the connected client, when 5 or more headers are received.

Conversely, in (B.2), submonitors are generated in a lazy manner, meaning that for each event message, only two submonitors are spawned. One of these submonitors detects a violation if the (http_eoh) message is received prematurely, otherwise it terminates. The other submonitor spawns two further submonitors if an HTTP information header is received. The property is reapplied if the end-of-header message is received after 5 or more information headers. Although (B.1) is more readable than (B.2), the latter is less resource intensive, as less submonitors are spawned for each connected client.

\[
\begin{align*}
\text{(B.2)}
\end{align*}
\]
B.1.2 Valid HTTP Request Format Property

In the HTTP protocol, every HTTP request must begin with a request notification header (ie, \texttt{http.req}). This request header must then be followed by a number of HTTP data headers (ie, \texttt{http.header}) which denote information about the requesting client and other type of information. Finally each request must end with an end-of-header notification (ie, \texttt{http.eoh}), denoting that the client has finished sending its request headers. The following property was therefore developed to make sure that every HTTP request received by the server follows this protocol.

\[
\max X. ( \quad \[\text{AcceptorPid!}[h\text{Pid},\text{next,}_.] \quad ( \quad [h\text{Pid} \supset \text{ret}([\text{yaws},\text{do_recv},3,[\text{ok,}[\text{http}\text{.req,GET,}_..]]])] \\
[\text{http.header}] X \\
) \quad \& \quad \text{(i) A data header message (ie, \texttt{http.header}) is received after a request header message (ie, \texttt{http.req}).} \\
) \quad \& \quad \text{(ii) Two data header messages are received subsequently.}
\]
B. Additional Properties and Tabulated Results

(iii) A data header message is followed by an end-of-header message (i.e., $\texttt{http\_eoh}$),
which is then followed by another request header message.

The property fails when a data header message is received \textit{before} a request header mes-
 sage, and also when a data header message is received \textit{after} an end-of-header notifica-
tion.

B.2 Detailed Result Tables

The following tables contain the same results plotted in the graphs given in Sec. 3.3.3 as Figures 3.6, 3.7 and 3.8. These results are structured in three subsections \textit{i.e.}, a subsection for each monitor synthesis algorithm \cite{39,15} (\textit{i.e.}, unoptimised, statically optimised and dynamically optimised) supported by detectEr (see Chapter A for details).

In each subsection we present: (i) the average CPU utilization ($\times10^6$ CPU cycles) per client request; (ii) the average memory consumption (MB) and (iii) the average response time per client request (ms); obtained when monitoring the Yaws webserver for a number properties using different monitoring approaches.

B.2.1 Results for Unoptimised Monitors

| Average CPU utilization per Request ($\times10^6$ CPU cycles) |
|---------------------------------------------------------------|
| Monitoring modes | Number of Requests |
| Baseline | 50 | 100 | 200 | 500 | 1000 | 2000 |
| Baseline | 14.556 | 14.408 | 14.127 | 14.171 | 14.365 | 14.269 |
| Asynchronous | 27.383 | 32.70 | 46.507 | 82.518 | 144.236 | 269.164 |
| Hybrid | 28.793 | 34.354 | 48.329 | 84.241 | 148.973 | 277.172 |
| Synchronous | 31.496 | 39.037 | 53.796 | 97.003 | 175.727 | 308.936 |
### Average Memory Consumption (MB)

| Monitoring modes | Number of Requests |
|------------------|--------------------|
|                  | 50  | 100 | 200 | 500 | 1000 | 2000 |
| Baseline         | 25.62 | 25.68 | 25.63 | 25.68 | 26.17 | 26.25 |
| Asynchronous     | 30.98 | 32.24 | 32.95 | 35.83 | 39.93 | 45.36 |
| Hybrid           | 31.41 | 32.39 | 33.38 | 36.26 | 41.02 | 49.02 |
| Synchronous      | 31.36 | 32.25 | 33.52 | 36.15 | 42.25 | 51.55 |

### Average response time per Request (ms)

| Monitoring modes | Number of Requests |
|------------------|--------------------|
|                  | 50  | 100 | 200 | 500 | 1000 | 2000 |
| Baseline         | 11.32 | 11.35 | 11.47 | 11.44 | 11.45 | 11.66 |
| Asynchronous     | 14.54 | 15.82 | 17.75 | 24.74 | 40.45 | 66.02 |
| Hybrid           | 17.30 | 17.331 | 19.75 | 27.30 | 44.82 | 75.62 |
| Synchronous      | 17.63 | 19.08 | 23  | 36.27 | 60.82 | 112.75 |

### Results for Statically Optimised Monitors

#### Average CPU utilization per Request ($\times 10^6$ CPU cycles)

| Monitoring modes | Number of Requests |
|------------------|--------------------|
|                  | 50  | 100 | 200 | 500 | 1000 | 2000 |
| Baseline         | 14.556 | 14.408 | 14.127 | 14.171 | 14.365 | 14.269 |
| Asynchronous     | 22.023 | 23.94 | 29.375 | 45.044 | 77.479 | 143.147 |
| Hybrid           | 22.75 | 24.468 | 30.35 | 45.284 | 78.177 | 144.179 |
| Synchronous      | 24.581 | 26.11 | 31.721 | 46.239 | 79.098 | 146.011 |
### B. Additional Properties and Tabulated Results

#### Average Memory Consumption (MB)

| Monitoring modes | 50   | 100  | 200  | 500  | 1000 | 2000 |
|------------------|------|------|------|------|------|------|
| Baseline         | 25.62| 25.68| 25.63| 25.68| 26.17| 26.25|
| Asynchronous     | 29.4 | 30.5 | 31.87| 33.79| 35.78| 40.09|
| Hybrid           | 29.73| 31.06| 32.25| 34.06| 36.2 | 41.15|
| Synchronous      | 30.08| 31.68| 32.24| 34.19| 36.11| 41.68|

#### Average response times per Request (ms)

| Monitoring modes | 50   | 100  | 200  | 500  | 1000 | 2000 |
|------------------|------|------|------|------|------|------|
| Baseline         | 11.32| 11.35| 11.47| 11.44| 11.45| 11.66|
| Asynchronous     | 13.66| 14.91| 16.31| 20.06| 27.29| 48.84|
| Hybrid           | 15.66| 15.94| 16.85| 21.41| 29.95| 53.08|
| Synchronous      | 18.07| 18.14| 19.61| 25.74| 39.11| 67.91|

#### B.2.3 Results for Dynamically Optimised Monitors

#### Average CPU utilization per Request ($\times 10^6$ CPU cycles)

| Monitoring modes | 50   | 100  | 200  | 500  | 1000 | 2000 |
|------------------|------|------|------|------|------|------|
| Baseline         | 14.556| 14.408| 14.127| 14.171| 14.365| 14.269|
| Asynchronous     | 20.175| 21.286| 23.732| 27.972| 33.909| 44.227|
| Hybrid           | 20.988| 21.516| 24.885| 29.755| 37.258| 50.191|
| Synchronous      | 23.369| 22.246| 26.272| 34.339| 46.773| 69.14|
### Average Memory Consumption (MB)

| Monitoring modes | 50   | 100  | 200  | 500  | 1000 | 2000 |
|------------------|------|------|------|------|------|------|
| Baseline         | 25.62| 25.68| 25.63| 25.68| 26.17| 26.25|
| Asynchronous     | 28.06| 28.56| 29.2 | 30.01| 30.68| 31.07|
| Hybrid           | 28.4 | 28.97| 29.69| 30.2 | 31.11| 31.95|
| Synchronous      | 28.88| 29.19| 29.62| 30.23| 31.33| 33.01|

### Average response times per Request (ms)

| Monitoring modes | 50   | 100  | 200  | 500  | 1000 | 2000 |
|------------------|------|------|------|------|------|------|
| Baseline         | 11.32| 11.35| 11.47| 11.44| 11.45| 11.66|
| Asynchronous     | 12.23| 12.74| 13.1 | 14.15| 16.18| 21.8 |
| Hybrid           | 13.11| 13.02| 13.31| 15.53| 18.39| 23.26|
| Synchronous      | 15.05| 15.014| 15.14| 16.74| 20.91| 29.68|
C. Proofs for Auxiliary Lemmas

We divide this chapter into two main parts. In the first part (Sec. C.1) we present the proof for Lemma C.1.1, an auxiliary lemma used when proving the Theorem of Semantic Correspondence (i.e., Thm. 4.5.1 in Sec. 4.5.1).

In the second part (Sec. C.2 and C.3) we present the proofs for the auxiliary lemmas required in proving the Theorem of Type Soundness (i.e., Thm. 5.2.1). We further divide this part into two sections: in Sec. C.2 we give the proof details about the primary auxiliary lemmas i.e., lemmas that are used directly by the main proofs presented in Sec. 5.3 whereas in Sec. C.3 we prove the secondary auxiliary lemmas i.e., lemmas that are used by the primary auxiliary lemmas. We now state the auxiliary lemmas that we prove in both parts of this chapter:

The Correspondence Auxiliary Lemma:

Lemma C.1.1 \( \phi \rightarrow \phi' \) and \( B, t \models \varphi' \) and \( A \Rightarrow B \) implies \( A, \gamma t \models \varphi \)

The Primary Soundness Auxiliary Lemmas:

Lemma C.2.1 (System Reduction)

\[ \Gamma' = \text{after}(\Gamma, \gamma) \text{ and } \text{wf}(\Gamma, s) \text{ and } s \gamma \rightarrow s' \text{ implies } \text{wf}(\Gamma', s') \]

Lemma C.2.2 (Monitor Reduction)

\[ \Sigma; \Gamma \vdash \varphi \text{ and } (\Gamma, \Delta, \varphi) \gamma \rightarrow (\Gamma', \Delta', \varphi') \text{ implies } \Sigma; \Gamma' \vdash \varphi' \text{ and } \Gamma' = \text{after}(\Gamma, \gamma) \]

Lemma C.2.3 (Formula Substitution)

\[ \Sigma; \Gamma' \vdash \text{max} X. \varphi \text{ and } \Sigma; \Gamma \vdash \varphi \text{ implies } \Sigma; \Gamma' \vdash [\text{clr}(X)\text{max} X. \varphi]/X] \]
C. Proofs for Auxiliary Lemmas

Lemma C.2.4 (Weakening) \( (\Sigma, X \mapsto \Gamma'); \Gamma \vdash \varphi \) and \( X \notin \text{fv}(\varphi) \) implies \( \Sigma; \Gamma \vdash \varphi \)

Lemma C.2.5 (Term Substitution)
\[
\Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi \quad \text{and} \quad \text{mtch}(e, \alpha) = \sigma \quad \text{and} \quad \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e) \sigma) = \emptyset
\]
and \( \forall (x, y): \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \) implies \( \Sigma; (\Gamma, \text{bnd}(e) \sigma) \vdash \varphi\sigma \)

The Secondary Soundness Auxiliary Lemmas:

Lemma C.3.1 \( \Sigma; \Gamma_1 \vdash \varphi \) and \( \Sigma; \Gamma_2 \vdash \psi \) and \( \Gamma = \Gamma_1 + \Gamma_2 \) implies \( \Sigma; \Gamma \vdash \varphi \& \psi \)

Lemma C.3.2 \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma', \Delta', \varphi' \rangle \) and \( \Sigma; \text{eff}(\Gamma, r) \vdash \varphi \) implies \( \langle \text{eff}(\Gamma, r), \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \text{eff}(\text{after}(\Gamma, \gamma), r), \Delta', \varphi' \rangle \) and \( \Gamma' = \text{after}(\Gamma, \gamma) \)

Lemma C.3.3 \( \Gamma = \Gamma_1 + \Gamma_2 \) and \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma', \Delta', \varphi' \rangle \) and \( \Sigma; \Gamma_1 \vdash \varphi \) implies \( \langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma'_1, \Delta', \varphi' \rangle \) and \( \Gamma' = \Gamma_1 + \Gamma_2 \)

Lemma C.3.4 \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma', \Delta', \varphi' \rangle \) implies \( \text{excl}(\varphi', \psi) = (r_\varphi, r_\psi) \)

Lemma C.3.5 \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma', \Delta', \varphi' \rangle \) and \( \Sigma; \text{eff}((\Gamma', \Gamma''), r_\psi) \vdash \varphi' \) and \( \Sigma; \text{eff}((\Gamma', \Gamma''), r_\psi) \vdash \psi' \) implies \( \Sigma; (\Gamma', \Gamma'') \vdash \varphi' \& \psi' \)

Lemma C.3.6 \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\cdot} \langle \Gamma', \Delta, \varphi' \rangle \) and \( \Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \psi \) and \( \Sigma; \text{eff}(\text{after}(\Gamma, \mu), r_\varphi) \vdash \psi' \) implies \( \Sigma; \text{after}(\Gamma, \mu) \vdash \varphi' \& \psi \)

Lemma C.3.7 \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \lceil (\text{clr}(X) \text{max } X, \varphi') / X \rceil \) implies \( \text{excl}(\varphi[\text{clr}(X) \text{max } X, \varphi') / X], \psi[\text{clr}(X) \text{max } X, \varphi') / X]) = (r_\varphi, r_\psi) \)

Lemma C.3.8 \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( \lceil (\text{clr}(X) \text{max } X, \varphi') / X \rceil \) implies \( \text{excl}(\varphi[\text{clr}(X) \text{max } X, \varphi') / X], \psi) = (r_\varphi, r_\psi) \)

Note that when proving the Primary and Secondary auxiliary lemmas for Type Soundness, we omit from showing the proofs for cases involving structural equivalence (i.e., \text{rStr}), idempotency (\text{rIdem1} and \text{rIdem2}) due to their triviality, as well as for cases that are analogous to other cases, i.e., their proof requires minimal effort to be derived from the proof of another case.
C. Proofs for Auxiliary Lemmas

C.1 Proving The Correspondence Auxiliary Lemma

To Prove Lemma C.1.1. \( \varphi \Rightarrow^\gamma \varphi' \) and \( B, t \models_v \varphi' \) and \( A \Rightarrow^\gamma B \) implies \( A, \gamma t \models_v \varphi \)

Proof. By rule induction on \( \varphi \Rightarrow^\gamma \varphi' \). The main cases are:

**Case rTru:** From our rule premises we know that if \( b \) then \( \varphi \) else \( \psi \rightarrow^\tau \varphi \) because

\[
b \Downarrow true \quad \text{(C.1)}
\]

and

\[
B, t \models_v \varphi \quad \text{(C.2)}
\]

\[
A \Rightarrow^\tau B \quad \text{(C.3)}
\]

By (C.1), (C.2), (C.3) and defn of \( \models_v \) we know \( A, \tau t \models_v \text{if } b \text{ then } \varphi \text{ else } \psi \) as required.

**Case rFls:** The proof for this case is analogous to that of rTru.

**Case rCn1:** From our rule premises we know that \( \varphi \& \psi \rightarrow^\alpha \varphi' \& \psi' \) because

\[
\varphi \rightarrow^\alpha \varphi' \quad \text{(C.4)}
\]

\[
\psi \rightarrow^\alpha \psi' \quad \text{(C.5)}
\]

and

\[
A \Rightarrow^\alpha B \quad \text{(C.6)}
\]

\[
B, t \models_v \varphi' \& \psi' \quad \text{(C.7)}
\]

By (C.7) and defn of \( \models_v \) we know

\[
B, t \models_v \varphi' \quad \text{(C.8)}
\]

or \( B, t \models_v \psi' \quad \text{(C.9)} \)

By (C.4), (C.6), (C.8) and IH we know

\[
A, \alpha t \models_v \varphi \quad \text{(C.10)}
\]

By (C.5), (C.6), (C.9) and IH we know

\[
A, \alpha t \models_v \psi \quad \text{(C.11)}
\]

By (C.10), (C.11) and defn of \( \models_v \) we know \( A, \alpha t \models_v \varphi \& \psi \) as required.
C. Proofs for Auxiliary Lemmas

Case rCn2: From our rule premises we know that $\varphi \land \psi \rightarrow \varphi' \land \psi$ because

$$\varphi \rightarrow \varphi'$$ \hspace{1cm} (C.12)

and

$$A \Rightarrow B$$ \hspace{1cm} (C.13)

$$B, t \vDash \varphi' \land \psi$$ \hspace{1cm} (C.14)

By (C.5) and defn of $\vDash$, we know

$$B, t \vDash \varphi'$$ \hspace{1cm} (C.15)

or

$$B, t \vDash \psi$$ \hspace{1cm} (C.16)

By (C.12), (C.13), (C.15) and IH we know

$$A, \tau t \vDash \varphi$$ \hspace{1cm} (C.17)

By (C.17) and the defn of $\vDash$, we know

$$A, \tau t \vDash \varphi' \land \psi$$ as required.

Case rMax: From our rule premises we know

$$\max X. \varphi \rightarrow \varphi[(\max X. \varphi)/X]$$ \hspace{1cm} (C.18)

$$A \Rightarrow B$$ \hspace{1cm} (C.19)

$$B, t \vDash \varphi[(\max X. \varphi)/X]$$ \hspace{1cm} (C.20)

By (C.20) and defn of $\vDash$, we know

$$B, t \vDash \max X. \varphi$$ \hspace{1cm} (C.21)

By (C.19) and (C.21) we know $A, \tau t \vDash \max X. \varphi$ as required.

Case rNc1: From our rule premises we know $[\alpha]\varphi \rightarrow \varphi\sigma$ because

$$\text{mtch}(e, \alpha) = \sigma$$ \hspace{1cm} (C.22)

and

$$A \overset{\alpha}{\Rightarrow} B$$ \hspace{1cm} (C.23)

$$B, t \vDash \varphi\sigma$$ \hspace{1cm} (C.24)

By (C.22), (C.23), (C.24) and defn of $\vDash$, we know $A, \alpha t \vDash [\alpha]\varphi$ as required.

Case rNc2: From our rule premises we know $[\alpha]\varphi \rightarrow \top$ because

$$\text{mtch}(e, \alpha) = \bot$$ \hspace{1cm} (C.25)
C. Proofs for Auxiliary Lemmas

and

\[ A \Rightarrow B \quad \text{(C.26)} \]
\[ B, \top \models \text{tt} \quad \text{(C.27)} \]

Since by the defn of \( \models \) we know that \( \text{(C.27)} \) can never be true, this means that we have a contradiction. \( \square \)

C.2 Proving The Primary Soundness Auxiliary Lemmas

To Prove Lemma C.2.1. (System Reduction)

\[ \Gamma' = \text{after}(\Gamma, \gamma) \quad \text{and} \quad \text{wf}(\Gamma, s) \quad \text{and} \quad s \Downarrow \gamma \implies \text{wf}(\Gamma', s'). \]

Proof. By Rule induction on \( s \Downarrow \gamma \).

Case sNew: From our rule premises we know

\[ s \Downarrow (s, i:\circ) \quad \text{(C.28)} \]
\[ \text{wf}(\Gamma, s) \quad \text{(C.29)} \]
\[ \text{after}(\Gamma, \tau) = \Gamma' \quad \text{(C.30)} \]

By (C.29) and System Well-Formedness (Def. 5.3.1) we know

\[ \Gamma(j) = \text{lbid \ implies} \ s(j) = \bullet \quad \text{(C.31)} \]

By (C.30) and the defn of after (Def. 5.3.2) and since \( \gamma = \tau \) we know that for any \( \varphi \)

\[ \Gamma' = \Gamma \quad \text{(C.32)} \]

Given that \( s \subseteq (s, i:\circ) \) and by (C.31) and (C.32) we know

\[ \Gamma'(j) = \text{lbid \ implies} \ (s, i:\circ)(j) = \bullet \quad \text{(C.33)} \]

By (C.33) and System Well-Formedness (Def. 5.3.1) we know

\[ \text{wf}(\Gamma', (s, i:\circ)) \quad \text{(C.34)} \]

\( \therefore \): Case holds by (C.34).

Case sAct: From our rule premises we know

\[ (s, i:\circ) \Downarrow (s, i:\circ) \quad \text{(C.35)} \]
because

\[ \text{subj}(\alpha) = i \]  \hspace{1cm} \text{(C.36)}
\[ \text{ids}(\alpha) \subseteq \text{dom}(s, i: \circ) \]  \hspace{1cm} \text{(C.37)}

and

\[ \text{wf}(\Gamma, (s, i: \circ)) \]  \hspace{1cm} \text{(C.38)}
\[ \text{after}(\Gamma, \alpha) = \Gamma' \]  \hspace{1cm} \text{(C.39)}

By (C.38) and System Well-Formedness (Def. 5.3.1) we know

\[ \Gamma(j) = \text{lbid} \text{ implies } (s, i: \circ)(j) = \bullet \]  \hspace{1cm} \text{(C.40)}

Since \( \gamma = \alpha \), by (C.39) and by defn of after (Def. 5.3.2) we know

\[ \Gamma \subseteq \Gamma' \]  \hspace{1cm} \text{(C.41)}

Recall that the type system is only able to change the type of a linear process id \( i: \text{lid} \) to internal type \( \text{lbid} \) whenever a \( \text{blk}(i) \) command is type-checked, so as to signify that the process reference should be blocked during runtime. This is because at runtime \( \text{blk}(i) \) generates a \( \text{b}(i) \) event which causes process id \( i \) to block. This means that an \( \alpha \) action can neither directly introduce a process \( j: \text{lbid} \) in the new new type environment \( \Gamma' \), nor block a process in state \( s \). Hence, from (C.40) and (C.41) we can deduce

\[ \Gamma'(j) = \text{lbid} \text{ implies } (s, i: \circ)(j) = \bullet \]  \hspace{1cm} \text{(C.42)}

By (C.42) and System Well-Formedness (Def. 5.3.1) we know

\[ \text{wf}(\Gamma', (s, i: \circ)) \]  \hspace{1cm} \text{(C.43)}

\( \therefore \) Case holds by (C.43).

**Case sBlk:** From our rule premises we know

\[ (s, w: \circ) \xrightarrow{\text{b}(w)} (s, w: \bullet) \]  \hspace{1cm} \text{(C.44)}
\[ \text{wf}(\Gamma, (s, w: \circ)) \]  \hspace{1cm} \text{(C.45)}
\[ \text{after}(\Gamma, \text{b}(w)) = \Gamma' \]  \hspace{1cm} \text{(C.46)}

By (C.46) the defn of after (Def. 5.3.2) and since \( \gamma = \text{b}(w) \) we know

\[ \Gamma = (\Gamma'', w: \text{lid}) \]  \hspace{1cm} \text{(C.47)}
\[ \Gamma' = (\Gamma'', w: \text{lbid}) \]  \hspace{1cm} \text{(C.48)}

By (C.45), (C.47) and System Well-Formedness (Def. 5.3.1) we know

\[ (\Gamma'', w: \text{lid})(j) = \text{lbid implies } (s, w: \circ)(j) = \bullet \]  \hspace{1cm} \text{(C.49)}
C. Proofs for Auxiliary Lemmas

From (C.48), (C.49) and since by (C.44) we know that \((s, w : \circ)\) becomes \((s, w : \bullet)\) we can deduce

\[(\Gamma'', w : \text{lid})(j) = \text{lid} \implies (s, w : \bullet)(j) = \bullet\]  \hspace{1cm} (C.50)

By (C.48), (C.50) and System Well-Formedness (Def. 5.3.1) we know

\[\text{wf}(\Gamma', (s, w : \bullet))\]  \hspace{1cm} (C.51)

∴ Case holds by (C.51).

**Case sRel:** From our rule premises we know

\[(s, w : \bullet) \xrightarrow{r(w)} (s, w : \circ)\]  \hspace{1cm} (C.52)

\[\text{wf}(\Gamma, (s, w : \bullet))\]  \hspace{1cm} (C.53)

\[\text{after}(\Gamma, r(w)) = \Gamma''\]  \hspace{1cm} (C.54)

By (C.54) the defn of after (Def. 5.3.2) and since \(\gamma = r(w)\) we know

\[\Gamma = (\Gamma'', w : \text{lid})\]  \hspace{1cm} (C.55)

\[\Gamma' = (\Gamma'', w : \text{lid})\]  \hspace{1cm} (C.56)

By (C.53), (C.55) and System Well-Formedness (Def. 5.3.1) we know

\[(\Gamma'', w : \text{lid})(j) = \text{lid} \implies (s, w : \bullet)(j) = \bullet\]  \hspace{1cm} (C.57)

From (C.56), (C.57) and since by (C.52) we know that \((s, w : \bullet)\) becomes \((s, w : \circ)\) we can deduce

\[(\Gamma'', w : \text{lid})(j) = \text{lid} \implies (s, w : \circ)(j) = \bullet\]  \hspace{1cm} (C.58)

By (C.56), (C.58) and System Well-Formedness (Def. 5.3.1) we know

\[\text{wf}(\Gamma', (s, w : \circ))\]  \hspace{1cm} (C.59)

∴ Case holds by (C.59).

**Case sAdS:** From our rule premises we know

\[(s, w : \bullet) \xrightarrow{s(w)} (s, w : \bullet)\]  \hspace{1cm} (C.60)

\[\text{wf}(\Gamma, (s, w : \bullet))\]  \hspace{1cm} (C.61)

\[\text{after}(\Gamma, s(w)) = \Gamma'\]  \hspace{1cm} (C.62)

By (C.62), the defn of after (Def. 5.3.2) and since \(\gamma = s(w)\)

\[\Gamma' = \Gamma\]  \hspace{1cm} (C.63)
By (C.61) and (C.63) we know
\[ \text{wf}(\Gamma', (s, w:\bullet)) \] (C.64)

∴ Case holds by (C.64).

**Case sAdA:** The proof for this case is analogous to that of case sAdS.

---

**To Prove Lemma C.2.2. (Monitor Reduction)**

\[ \Sigma; \Gamma \vdash \varphi \quad \text{and} \quad \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \text{ implies } \Sigma; \Gamma' \vdash \varphi' \text{ and } \Gamma' = \text{after}(\Gamma, \gamma). \]

**Proof:** By Rule induction on \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \).

**Note:** This is a continuation of the proof given in Sec. 5.3.5

**Case rTru:** From our rule premises we know
\[ \langle \Gamma, \Delta, \text{if } c \text{ then } \varphi \text{ else } \psi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta, \varphi \rangle \quad \text{(because } c \Downarrow \text{true)} \] (C.65)
\[ \Sigma; \Gamma \vdash \text{if } c \text{ then } \varphi \text{ else } \psi \] (C.66)

By (C.66) and tIf we know
\[ \Sigma; \Gamma \vdash \varphi \] (C.67)
\[ \Sigma; \Gamma \vdash \psi \] (C.68)

By the defn of after (Def. 5.3.2) and since \( \gamma = \tau \) we know
\[ \Gamma = \text{after}(\Gamma, \tau) \] (C.69)

∴ Case holds by (C.67) and (C.69).

**Note:** The proof for cases rFls and rClr are analogous to the proof for case rTru.

**Case rCn1:** From our rule premises we know
\[ \langle \Gamma, \Delta, \varphi \& \psi \rangle \xrightarrow{\alpha} \langle (\Gamma', \Gamma''), (\Delta' \cup \Delta''), \varphi' \& \psi' \rangle \] (C.70)

because
\[ \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle \] (C.71)
\[ \langle \Gamma, \Delta, \psi \rangle \xrightarrow{\alpha} \langle \Gamma'', \Delta'', \psi' \rangle \] (C.72)
\[ \text{dom}(\Delta) = \text{dom}(\Delta') \cap \text{dom}(\Delta'') \] (C.73)
C. Proofs for Auxiliary Lemmas

and

$$\Sigma; \Gamma \vdash \psi$$ \hspace{1cm} (C.74)

As $\varphi$ and $\psi$ may either be exclusive or not, we must consider the following two subcases:

**Subcase** $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$: Since $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$, by (C.74) and $\tau \text{C} n 2$ we know

$$\Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \varphi$$ \hspace{1cm} (C.75)

$$\Sigma; \text{eff}(\Gamma, r_\psi) \vdash \psi$$ \hspace{1cm} (C.76)

By (C.71), (C.75) and Lemma C.3.2 we know

$$\langle \text{eff}(\Gamma, r_\varphi), \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \text{eff}(\text{after}(\Gamma, \alpha), r_\varphi), \Delta', \varphi' \rangle$$ \hspace{1cm} (C.77)

$$\Gamma' = \text{after}(\Gamma, \alpha)$$ \hspace{1cm} (C.78)

By (C.72), (C.76) and Lemma C.3.2 we know

$$\langle \text{eff}(\Gamma, r_\psi), \Delta, \psi \rangle \xrightarrow{\alpha} \langle \text{eff}(\text{after}(\Gamma, \alpha), r_\psi), \Delta'', \psi' \rangle$$ \hspace{1cm} (C.79)

$$\Gamma'' = \text{after}(\Gamma, \alpha)$$ \hspace{1cm} (C.80)

By (C.75), (C.77) and IH we know

$$\Sigma; \text{eff}(\text{after}(\Gamma, \alpha), r_\varphi) \vdash \varphi'$$ \hspace{1cm} (C.81)

after$(\text{eff}(\Gamma, r_\varphi), \alpha) = \text{eff}(\text{after}(\Gamma, \alpha), r_\varphi)$ \hspace{1cm} (C.82)

By (C.76), (C.79) and IH we know

$$\Sigma; \text{eff}(\text{after}(\Gamma, \alpha), r_\psi) \vdash \psi'$$ \hspace{1cm} (C.83)

after$(\text{eff}(\Gamma, r_\psi), \alpha) = \text{eff}(\text{after}(\Gamma, \alpha), r_\psi)$ \hspace{1cm} (C.84)

By (C.82), (C.84) and the defn of after (Def. 5.3.2) we know

$$\text{eff}(\Gamma, r_\varphi) \subseteq \text{eff}(\text{after}(\Gamma, \alpha), r_\varphi)$$ \hspace{1cm} (C.85)

$$\text{eff}(\Gamma, r_\psi) \subseteq \text{eff}(\text{after}(\Gamma, \alpha), r_\psi)$$ \hspace{1cm} (C.86)

Hence from (C.78), (C.80), (C.85) and (C.86) we can deduce

$$\Gamma \subseteq (\Gamma', \Gamma'')$$ \hspace{1cm} (C.87)

By (C.87) and the defn of after (Def. 5.3.2) we can deduce

$$\text{after}(\Gamma, \alpha) = (\Gamma', \Gamma'')$$ \hspace{1cm} (C.88)

By (C.81), (C.83) and (C.88) we can deduce

$$\Sigma; \text{eff}((\Gamma', \Gamma''), r_\varphi) \vdash \varphi'$$ \hspace{1cm} (C.89)

$$\Sigma; \text{eff}((\Gamma', \Gamma''), r_\psi) \vdash \psi'$$ \hspace{1cm} (C.90)
C. Proofs for Auxiliary Lemmas

By (C.73) we know that we do not have any type incompatibilities, and since $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$, by (C.71), (C.89), (C.90) and Lemma C.3.5 we know

$$\Sigma; (\Gamma', \Gamma'') \vdash \varphi' \& \psi' \quad (C.91)$$

$\therefore$ Subcase holds by (C.88) and (C.91).

**Subcase** $\text{excl}(\varphi, \psi) = \bot$: Since $\text{excl}(\varphi, \psi) = \bot$, by (C.74) and tCN1 we know

$$\Gamma = (\Gamma_1 + \Gamma_2) \quad (C.92)$$

$$\Sigma; \Gamma_1 \vdash \varphi \quad (C.93)$$

$$\Sigma; \Gamma_2 \vdash \psi \quad (C.94)$$

By (C.71), (C.92), (C.93) and Lemma C.3.3 we know

$$\langle \Gamma_1, \Delta, \varphi \rangle \overset{\alpha}{\rightarrow} \langle \Gamma'_1, \Delta', \varphi' \rangle \quad (C.95)$$

$$\Gamma' = (\Gamma_1 + \Gamma_2) \quad (C.96)$$

By (C.72), (C.92), (C.94) and Lemma C.3.3 we know

$$\langle \Gamma_2, \Delta, \varphi \rangle \overset{\alpha}{\rightarrow} \langle \Gamma'_2, \Delta', \varphi' \rangle \quad (C.97)$$

$$\Gamma'' = (\Gamma_1 + \Gamma_2) \quad (C.98)$$

By (C.93), (C.95) and IH we know

$$\Sigma; \Gamma'_1 \vdash \varphi' \quad (C.99)$$

$$\Gamma'_1 = \text{after}(\Gamma_1, \alpha) \quad (C.100)$$

By (C.94), (C.97) and IH we know

$$\Sigma; \Gamma'_2 \vdash \psi' \quad (C.101)$$

$$\Gamma'_2 = \text{after}(\Gamma_2, \alpha) \quad (C.102)$$

By (C.100), (C.102) and the defn of after (Def. 5.3.2) we know

$$\Gamma_1 \subseteq \Gamma'_1 \quad (C.103)$$

$$\Gamma_2 \subseteq \Gamma'_2 \quad (C.104)$$

Since by (C.73) we know that we do not have aliasing, then by (C.96), (C.98), (C.103) and (C.104) we can deduce

$$(\Gamma', \Gamma'') = (\Gamma'_1 + \Gamma'_2) \quad (C.105)$$

By (C.99), (C.101), (C.105) and Lemma C.3.1 we know

$$\Sigma; (\Gamma', \Gamma'') \vdash \varphi' \& \psi' \quad (C.106)$$
By (C.92), (C.96), (C.98), (C.103) and (C.104) we can deduce
\[ \Gamma \subseteq (\Gamma', \Gamma'') \] (C.107)

By (C.107) and the definition of after (Def. 5.3.2) we know
\[ (\Gamma', \Gamma'') = \text{after}(\Gamma, \alpha) \] (C.108)

\[ \therefore \text{Subcase holds by (C.106) and (C.108)}. \]

**Case rCn2:** From our rule premises we know
\[ \langle \Gamma, \Delta, \varphi & \psi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta', \varphi' & \psi \rangle \] (C.109)

because
\[ \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta', \varphi' \rangle \] (C.110)

and
\[ \Sigma; \Gamma \vdash \varphi & \psi \] (C.111)

As \( \varphi \) and \( \psi \) may either be exclusive or not, we must consider the following two subcases:

**Subcase excl(\( \varphi, \psi \)) = (r_\varphi, r_\psi):** Since \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \), by (C.111) and rCn2 we know
\[ \Sigma; \text{eff}(\Gamma, r_\psi) \vdash \varphi \] (C.112)
\[ \Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \psi \] (C.113)

By (C.110), (C.112) and Lemma C.3.2 we know
\[ \langle \text{eff}(\Gamma, r_\psi), \Delta, \varphi \rangle \xrightarrow{\tau} \langle \text{eff}(\Gamma, r_\psi), \Delta', \varphi' \rangle \] (C.114)

By (C.112), (C.114) and IH we know
\[ \Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \varphi' \] (C.115)

Since \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \), by (C.114) and Lemma C.3.4 we can deduce
\( \text{excl}(\varphi', \psi) = (r_\varphi, r_\psi) \) (C.116)

By (C.113), (C.115), (C.116) and rCn2 we know
\[ \Sigma; \Gamma \vdash \varphi' & \psi \] (C.117)

By the definition of after (Def. 5.3.2) we know
\[ \Gamma = \text{after}(\Gamma, \tau) \] (C.118)

\[ \therefore \text{Subcase holds by (C.117) and (C.118)}. \]
Subcase $\text{excl}(\varphi, \psi) = \bot$: By (C.111) and $\tau \text{Cn} 1$ we know

\[
\Gamma = (\Gamma_1 + \Gamma_2) \\
\Sigma; \Gamma_1 \vdash \varphi \\
\Sigma; \Gamma_2 \vdash \psi
\]  

(C.119) 

(C.120) 

(C.121)

By (C.110), (C.119), (C.120) and Lemma [C.3.3] we know

\[
\langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{r} \langle \Gamma_1, \Delta', \varphi' \rangle
\]

(C.122)

By (C.120), (C.122) and IH we know

\[
\Sigma; \Gamma_1 \vdash \varphi'
\]

(C.123)

\[
\Gamma_1 = \text{after}(\Gamma_1, \tau)
\]

(C.124)

By (C.119), (C.121), (C.123) and Lemma [C.3.1] we know

\[
\Sigma; \Gamma \vdash \varphi' \& \psi
\]

(C.125)

By the defn of after (Def. 5.3.2) we know

\[
\Gamma = \text{after}(\Gamma, \tau)
\]

(C.126)

∴ Subcase holds by (C.125) and (C.126).

Case $\text{rCn} 3$: From our rule premises we know

\[
\langle \Gamma, \Delta, \varphi \& \psi \rangle \xrightarrow{\mu} \langle \Gamma', \Delta, \varphi' \& \psi \rangle
\]

(C.127)

because

\[
\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\mu} \langle \Gamma', \Delta, \varphi' \rangle
\]

(C.128)

and

\[
\Sigma; \Gamma \vdash \varphi \& \psi
\]

(C.129)

As $\varphi$ and $\psi$ may either be exclusive or not, we must consider the following two subcases:

Subcase $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$: Since $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$, by (C.129) and $\tau \text{Cn} 2$ we know

\[
\Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \varphi
\]

(C.130)

\[
\Sigma; \text{eff}(\Gamma, r_\psi) \vdash \psi
\]

(C.131)
By (C.128), (C.130) and Lemma C.3.2 we know
\[ \langle \text{eff}(\Gamma, r_\psi), \Delta, \varphi \rangle \xrightarrow{\mu} \langle \text{eff}(\text{after}(\Gamma, \mu), r_\psi), \Delta, \varphi' \rangle \] (C.132)
\[ \Gamma' = \text{after}(\Gamma, \mu) \] (C.133)

By (C.130), (C.132) and IH we know
\[ \Sigma; \text{eff}(\text{after}(\Gamma, \mu), r_\psi) \vdash \varphi' \] (C.134)
\[ \text{after}(\text{eff}(\Gamma, r_\psi), \mu) = \text{eff}(\text{after}(\Gamma, \mu), r_\psi) \] (C.135)

Since excl(\varphi, \psi) = (r_\varphi, r_\psi), by (C.128), (C.131), (C.134) and Lemma C.3.6 we know
\[ \Sigma; \text{after}(\Gamma, \mu) \vdash \varphi' \& \psi \] (C.136)
\[ \therefore \text{Subcase holds by (C.133) and (C.136)}. \]

**Subcase** excl(\varphi, \psi) = ⊥: By (C.129) and τCν1 we know
\[ \Gamma = (\Gamma_1 + \Gamma_2) \] (C.137)
\[ \Sigma; \Gamma_1 \vdash \varphi \] (C.138)
\[ \Sigma; \Gamma_2 \vdash \psi \] (C.139)

By (C.128), (C.137), (C.138) and Lemma C.3.3 we know
\[ \langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\mu} \langle \Gamma'_1, \Delta, \varphi' \rangle \] (C.140)
\[ \Gamma' = (\Gamma'_1 + \Gamma_2) \] (C.141)

By (C.138), (C.140) and IH we know
\[ \Sigma; \Gamma'_1 \vdash \varphi' \] (C.142)
\[ \Gamma'_1 = \text{after}(\Gamma_1, \mu) \] (C.143)

By (C.139), (C.141), (C.142) and Lemma C.3.1 we know
\[ \Sigma; \Gamma' \vdash \varphi' \& \psi \] (C.144)

By (C.137), (C.141) and (C.143) we deduce
\[ (\Gamma'_1 + \Gamma_2 = \text{after}(\Gamma_1, \mu) + \Gamma_2) \equiv (\Gamma' = \text{after}(\Gamma, \mu)) \] (C.145)
\[ \therefore \text{Subcase holds by (C.144) and (C.145)}. \]

**Case** rRel: From our rule premises we know
\[ \langle (\Gamma, w : \text{lid}), \Delta, \text{rel}(w) \varphi \rangle \xrightarrow{r(n)} \langle (\Gamma, w : \text{lid}), \Delta, \varphi \rangle \] (C.146)
\[ \Sigma; (\Gamma, w : \text{lid}) \vdash \text{rel}(w) \varphi \] (C.147)

175
By (C.147) and \( \tau \text{REL} \) we know
\[
\Sigma; (\Gamma, w: \text{lid}) \vdash \varphi
\]  
(C.148)

By the defn of \( \text{after} \) (Def. 5.3.2) and since \( \gamma = r(w) \) we know
\[
\text{after}((\Gamma, w: \text{lid}), r(w)) = (\Gamma, w: \text{lid})
\]  
(C.149)

\( \therefore \) Case holds by (C.148) and (C.149).

**Case rBLK:** From our rule premises we know
\[
\langle (\Gamma, w: \text{lid}), \Delta, \text{blk}(w) \varphi \rangle \xrightarrow{\text{b}(w)} \langle (\Gamma, w: \text{lid}), \Delta, \varphi \rangle
\]  
(C.150)
\[
\Sigma; (\Gamma, w: \text{lid}) \vdash \text{blk}(w) \varphi
\]  
(C.151)

By (C.151) and \( \tau \text{BLK} \) we know
\[
\Sigma; (\Gamma, w: \text{lid}) \vdash \varphi
\]  
(C.152)

By the defn of \( \text{after} \) (Def. 5.3.2) and since \( \gamma = b(w) \) we know
\[
\text{after}((\Gamma, w: \text{lid}), b(w)) = (\Gamma, w: \text{lid})
\]  
(C.153)

\( \therefore \) Case holds by (C.152) and (C.153).

**Case rAda:** From our rule premises we know
\[
\langle \Gamma, \Delta, aA(w), \varphi \rangle \xrightarrow{a(w)} \langle \Gamma, \Delta, \text{rel}(r) \varphi \rangle
\]  
(C.154)
\[
\Sigma; \Gamma \vdash aA(w), \varphi
\]  
(C.155)

By (C.155) and \( \tau \text{Ada} \) we know
\[
\Gamma = (\Gamma', w: \text{lid})
\]  
(C.156)
\[
\Sigma; \Gamma \vdash \text{rel}(r) \varphi
\]  
(C.157)

By the defn of \( \text{after} \) (Def. 5.3.2) and since \( \gamma = a(w) \) we know
\[
\text{after}(\Gamma, a(w)) = \Gamma
\]  
(C.158)

\( \therefore \) Case holds by (C.157) and (C.158).

**Case rAdS:** The proof for this case is analogous to the proof of case rAda.

**Case rNc3:** From our rule premises we know
\[
\langle \Gamma, \Delta, [e_x]^\alpha \varphi \rangle \xrightarrow{\alpha} \langle \Gamma, \Delta, \text{rel}(r) \text{tt} \rangle
\]  
(C.159)
because

\[ \text{mtch}(e, \alpha) = \bot \]  \hspace{1cm} (C.160)

and

\[ \Sigma; \Gamma \vdash [e]^{\rho}_{\varphi} \]  \hspace{1cm} (C.161)

From (C.159) we know that \([e]^{\rho}_{\varphi}\) reduces to

\[ \text{rel}(r) \text{tt} \]  \hspace{1cm} (C.162)

Since \( \gamma = \alpha \), by the defn of after (Def. 5.3.2) we know

\[ \Gamma \subseteq \text{after}(\Gamma, \alpha) \]  \hspace{1cm} (C.163)

By (C.159) we know that \( \Gamma \) is not modified by the reduction, and hence from (C.163) we can deduce

\[ \Gamma = \text{after}(\Gamma, \alpha) \]  \hspace{1cm} (C.164)

As \( \rho \in \{b, a\} \) we must consider two subcases:

**Subcase** \( \rho = 'b': \) By (C.161) and \( \tau \text{NcB} \) we know

\[ \Sigma; \Gamma \vdash \text{rel}(r) \text{tt} \]  \hspace{1cm} (C.165)

\( \therefore \) Subcase holds by (C.164) and (C.165).

**Subcase** \( \rho = 'a': \) The proof for this subcase is analogous to that of the previous subcase.

\[ \square \]

**To Prove Lemma C.2.3. (Formula Substitution)**

\( (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \psi \) and \( \Sigma; \Gamma' \vdash \max X \varphi \) implies \( (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \psi[(\text{clr}(X)\max X \varphi)/X] \).

**Proof.** By rule induction on \( (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \psi \).

**Case** \( \tau \text{FLS}: \) From our rule premises we know

\[ (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \text{ff} \]  \hspace{1cm} (C.166)

\[ \Sigma; \Gamma' \vdash \max X \varphi \]  \hspace{1cm} (C.167)

Since \( \text{ff} \not= X \), by substitution we know

\[ \text{ff}[(\text{clr}(X)\max X \varphi)/X] \equiv \text{ff} \]  \hspace{1cm} (C.168)

By (C.166) and (C.168) we know

\[ (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \text{ff}[(\text{clr}(X)\max X \varphi)/X] \]  \hspace{1cm} (C.169)
C. Proofs for Auxiliary Lemmas

∴ Case holds by (C.169).

Note: The proof for case τTrue is analogous to the proof for case τFalse.

Case τVar: We must consider two subcases, one where the formula variable Y is equal to X (i.e., the substitution is applied) and another one where Y is not equal to X.

Subcase Y = X: From our premises we know

\[(\Sigma; \{X \mapsto \Gamma\}); \Gamma \vdash X\] (C.170)
\[\Sigma; \Gamma' \vdash \max X \varphi\] (C.171)

Since Y = X and by substitution we know

\[X[(\text{clr}(X)\max X \varphi)/X] \equiv (\text{clr}(X)\max X \varphi)\] (C.172)

By (C.170) and τVar we know

\[\Sigma(X) \subseteq \Gamma \equiv \Gamma' \subseteq \Gamma\] (C.173)

By (C.171), (C.173) and transitivity we know

\[\Sigma; \Gamma' \vdash \max X \varphi\] (C.174)

By (C.174) and τClr we know

\[\Sigma; \Gamma' \vdash \text{clr}(X)\max X \varphi\] (C.175)

As we work up-to α-equivalence we know that Σ does not contain an entry for formula variable X. In light of this, from (C.175) we can deduce

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{clr}(X)\max X \varphi\] (C.176)

By (C.172) and (C.176) we know

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash X[(\text{clr}(X)\max X \varphi)/X]\] (C.177)

∴ Subcase holds by (C.177).

Subcase Y ≠ X: From our premises we know

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash Y\] (C.178)
\[\Sigma; \Gamma' \vdash \max X \varphi\] (C.179)

Since Y ≠ X, by substitution we know

\[Y[(\text{clr}(X)\max X \varphi)/X] \equiv Y\] (C.180)

178
C. Proofs for Auxiliary Lemmas

By (C.178) and (C.180) we know
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash Y[(\text{clr}(X)\text{max} X. \varphi)/X]
\] (C.181)
∴ Subcase holds by (C.181).

**Case τMAX:** From our rule premises we know
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{max} Y. \psi
\] (C.182)
because
\[
\Sigma, \{X \mapsto \Gamma', Y \mapsto \Gamma\}; \Gamma \vdash \psi
\] (C.183)
and
\[
\Sigma; \Gamma' \vdash \text{max} X. \varphi
\] (C.184)
By substitution we know
\[
\text{max} Y. \psi)[(\text{clr}(X)\text{max} X. \varphi)/X] \equiv \text{max} Y. (\psi[(\text{clr}(X)\text{max} X. \varphi)/X])
\] (C.185)
By (C.183), (C.184) and IH we know
\[
\Sigma, \{X \mapsto \Gamma', Y \mapsto \Gamma\}; \Gamma \vdash \psi[(\text{clr}(X)\text{max} X. \varphi)/X]
\] (C.186)
By (C.186) and τMAX we know
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{max} Y. (\psi[(\text{clr}(X)\text{max} X. \varphi)/X])
\] (C.187)
By (C.185) and (C.187) we know
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{max} Y. \psi[(\text{clr}(X)\text{max} X. \varphi)/X]
\] (C.188)
∴ Case holds by (C.188).

**Case τCLR:** The proof for this case is analogous to that of case τMAX

**Case τAdS:** From our rule premises we know
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{sA}(w), \psi
\] (C.189)
because
\[
\Gamma = \Gamma'', w: \text{lbid}
\] (C.190)
\[
(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{rel}(r) \psi
\] (C.191)
C. Proofs for Auxiliary Lemmas

and

\[ \Sigma; \Gamma' \vdash \max X. \varphi \]  \hspace{1cm} (C.192)

By substitution we know

\[ sA(w), (\psi[\max X. \varphi]/X) \equiv (sA(w), \psi)[\max X. \varphi]/X \]  \hspace{1cm} (C.193)

By (C.191), (C.192) and IH we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \text{rel}(r) \psi[\max X. \varphi]/X \]  \hspace{1cm} (C.194)

By (C.190), (C.194) and \text{taAdS} we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash sA(w), (\psi[\max X. \varphi]/X) \]  \hspace{1cm} (C.195)

By (C.193) and (C.195) we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash (sA(w), \psi)[\max X. \varphi]/X \]  \hspace{1cm} (C.196)

∴ Case holds by (C.196).

\textbf{Case \text{taAdA}:} The proof for this case is analogous to the proof of cases \text{taAdS}.

\textbf{Case \text{taIf}:} From our rule premises we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \text{if } c \text{ then } \psi_1 \text{ else } \psi_2 \]  \hspace{1cm} (C.197)

because

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \psi_1 \]  \hspace{1cm} (C.198)

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \psi_2 \]  \hspace{1cm} (C.199)

and

\[ \Sigma; \Gamma' \vdash \max X. \varphi \]  \hspace{1cm} (C.200)

By substitution we know

\[ \text{if } c \text{ then } \psi_1 \text{ else } \psi_2)[\max X. \varphi]/X \] \hspace{1cm} \equiv \hspace{1cm} \text{if } c \text{ then } (\psi_1)[\max X. \varphi]/X \text{ else } (\psi_2)[\max X. \varphi]/X \]  \hspace{1cm} (C.201)

By (C.198), (C.200) and IH we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \psi_1[\max X. \varphi]/X \]  \hspace{1cm} (C.202)

By (C.199), (C.200) and IH we know

\[ (\Sigma, \{ X \mapsto \Gamma' \}); \Gamma \vdash \psi_2[\max X. \varphi]/X \]  \hspace{1cm} (C.203)

180
By (C.202), (C.203) and \( \Gamma \vdash \text{if } c \text{ then } (\psi_1[(\text{clr}(X) \max X. \varphi)/X]) \text{ else } (\psi_2[(\text{clr}(X) \max X. \varphi)/X]) \) (C.204)

By (C.201) and (C.204) we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma \vdash (\text{if } c \text{ then } \psi_1 \text{ else } \psi_2)[(\text{clr}(X) \max X. \varphi)/X]
\] (C.205)

\[\therefore \text{ Case holds by (C.205)}.\]

**Case \( \Gamma_{cn1} \):** From our rule premises we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma \vdash \psi_1 \& \psi_2
\] (C.206)

because
\[
\Gamma = (\Gamma_1 + \Gamma_2) \quad \text{(C.207)}
\]
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma_1 \vdash \psi_1 \quad \text{(C.208)}
\]
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma_2 \vdash \psi_2 \quad \text{(C.209)}
\]

and
\[
\Sigma; \Gamma_1 \vdash \text{max } X. \varphi \quad \text{(C.210)}
\]

By substitution we know
\[
(\psi_1 \& \psi_2)[(\text{clr}(X) \max X. \varphi)/X] \equiv (\psi_1[(\text{clr}(X) \max X. \varphi)/X]) \& (\psi_2[(\text{clr}(X) \max X. \varphi)/X])
\] (C.212)

By (C.208), (C.211) and IH we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma_1 \vdash (\psi_1[(\text{clr}(X) \max X. \varphi)/X])
\] (C.213)

By (C.209), (C.211) and IH we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma_2 \vdash (\psi_2[(\text{clr}(X) \max X. \varphi)/X])
\] (C.214)

By (C.207), (C.213), (C.214) and Lemma [C.3.1] we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); (\Gamma_1 + \Gamma_2) \vdash (\psi_1[(\text{clr}(X) \max X. \varphi)/X]) \& (\psi_2[(\text{clr}(X) \max X. \varphi)/X])
\] (C.215)

By (C.207), (C.212) and (C.215) we know
\[
(\Sigma; \{X \mapsto \rightarrow \Gamma\}); \Gamma \vdash (\psi_1 \& \psi_2)[(\text{clr}(X) \max X. \varphi)/X]
\] (C.216)

\[\therefore \text{ Case holds by (C.216)}.\]
C. Proofs for Auxiliary Lemmas

**Case tCn2:** From our rule premises we know

\[(\Sigma,\{X \mapsto \Gamma\}); \Gamma \vdash \psi_1 \& \psi_2\]  \hspace{1cm} \text{(C.217)}

because

\[\text{excl}(\psi_1, \psi_2) = (r_1, r_2)\]  \hspace{1cm} \text{(C.218)}

\[(\Sigma,\{X \mapsto \Gamma\}); \text{eff}(\Gamma, r_2) \vdash \psi_1\]  \hspace{1cm} \text{(C.219)}

\[(\Sigma,\{X \mapsto \Gamma\}); \text{eff}(\Gamma, r_1) \vdash \psi_2\]  \hspace{1cm} \text{(C.220)}

and

\[\Sigma; \Gamma' \vdash \text{max } X. \varphi\]  \hspace{1cm} \text{(C.221)}

By substitution we know

\[(\psi_1 \& \psi_2)[(\text{clr}(X)\text{max } X. \varphi)/X] \equiv (\psi_1[(\text{clr}(X)\text{max } X. \varphi)/X]) \& (\psi_2[(\text{clr}(X)\text{max } X. \varphi)/X])\]  \hspace{1cm} \text{(C.222)}

By (C.219), (C.221) and IH we know

\[(\Sigma,\{X \mapsto \Gamma\}); \text{eff}(\Gamma, r_2) \vdash \psi_1[(\text{clr}(X)\text{max } X. \varphi)/X]\]  \hspace{1cm} \text{(C.223)}

By (C.220), (C.221) and IH we know

\[(\Sigma,\{X \mapsto \Gamma\}); \text{eff}(\Gamma, r_1) \vdash \psi_2[(\text{clr}(X)\text{max } X. \varphi)/X]\]  \hspace{1cm} \text{(C.224)}

Since \[(\text{clr}(X)\text{max } X. \varphi)/X],\] by (C.218) and Lemma C.3.7 we know

\[\text{excl}(\psi_1[(\text{clr}(X)\text{max } X. \varphi)/X], \psi_2[(\text{clr}(X)\text{max } X. \varphi)/X]) = (r_1, r_2)\]  \hspace{1cm} \text{(C.225)}

By (C.223), (C.224), (C.225) and tCn2 we know

\[(\Sigma,\{X \mapsto \Gamma\}); \Gamma \vdash (\psi_1[(\text{clr}(X)\text{max } X. \varphi)/X]) \& (\psi_2[(\text{clr}(X)\text{max } X. \varphi)/X])\]  \hspace{1cm} \text{(C.226)}

From (C.222) and (C.226) we can deduce

\[(\Sigma,\{X \mapsto \Gamma\}); \Gamma \vdash (\psi_1 \& \psi_2)[(\text{clr}(X)\text{max } X. \varphi)/X]\]  \hspace{1cm} \text{(C.227)}

\[\therefore \text{Case holds by (C.227).}\]

**Case tNcB:** From our rule premises we know

\[(\Sigma,\{X \mapsto \Gamma\}); \Gamma \vdash [e]_b^b \psi\]  \hspace{1cm} \text{(C.228)}

because

\[\text{subj}(e) = l\]  \hspace{1cm} \text{(C.229)}

\[(\Sigma,\{X \mapsto \Gamma\}); (\Gamma, \text{bnd}(e)) \vdash \text{blk}(l) \psi\]  \hspace{1cm} \text{(C.230)}

\[\Sigma; \Gamma \vdash \text{rel}(r) \text{tt}\]  \hspace{1cm} \text{(C.231)}
C. Proofs for Auxiliary Lemmas

and

\[ \Sigma; \Gamma' \vdash \text{max } X \varphi \]  \hspace{1cm} (C.232)

By substitution we know

\[ ([e]_\varphi^b)(\text{clr}(X)\text{max } X \varphi)/X] = [e]_\varphi^b (\psi[\text{clr}(X)\text{max } X \varphi])/X \]  \hspace{1cm} (C.233)

By (C.230), (C.232) and IH we know

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}; \Gamma, \text{bnd}(e)) \vdash \text{blk}(\text{l}\psi)[\text{clr}(X)\text{max } X \varphi]/X \]  \hspace{1cm} (C.234)

By substitution we also know

\[ \text{blk}(\text{w}\psi)[\text{clr}(X)\text{max } X \varphi]/X] \equiv \text{blk}(\text{l}\psi)[\text{clr}(X)\text{max } X \varphi]/X \]  \hspace{1cm} (C.235)

By (C.234) and (C.235) we know

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}; \Gamma, \text{bnd}(e)) \vdash [e]_\varphi^b \psi[\text{clr}(X)\text{max } X \varphi]/X \]  \hspace{1cm} (C.236)

By (C.229), (C.231), (C.236) \( \tau \text{NCB} \) we know

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}; \Gamma, \text{bnd}(e)) \vdash \text{rel}(\text{w}\psi) \]  \hspace{1cm} (C.237)

By (C.233) and (C.237) we know

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}; \Gamma, \text{bnd}(e)) \vdash \text{rel}(\text{w}\psi)[\text{clr}(X)\text{max } X \varphi]/X \]  \hspace{1cm} (C.238)

\[ \therefore \text{ Case holds by (C.238).} \]

**Case \( \tau \text{NCB} \):** The proof for this case is analogous to that of case \( \tau \text{NCB} \).

**Case \( \tau \text{REL} \):** From our rule premises we know

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \text{rel}(\text{w}\psi) \]  \hspace{1cm} (C.239)

because

\[ \Gamma = (\Gamma'', \text{w}: \text{lbid}) \]  \hspace{1cm} (C.240)

\[ \text{(} \Sigma, \{X \mapsto \Gamma'\}); \Gamma'' \vdash \text{lid}\psi \]  \hspace{1cm} (C.241)

and

\[ \Sigma; \Gamma' \vdash \text{max } X \varphi \]  \hspace{1cm} (C.242)

By substitution we know

\[ \text{rel}(\text{w}) (\psi[\text{clr}(X)\text{max } X \varphi]/X) \equiv \text{rel}(\text{w}) (\psi)[\text{clr}(X)\text{max } X \varphi]/X \]  \hspace{1cm} (C.243)
C. Proofs for Auxiliary Lemmas

By (C.241), (C.242) and IH we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma''', w : \text{lid} \vdash \psi[(\text{clr}(X) \text{max } X, \varphi)/X] \quad (C.244)\]

By (C.240), (C.244) and tRel we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{rel}(w)(\psi[(\text{clr}(X) \text{max } X, \varphi)/X]) \quad (C.245)\]

By (C.243) and (C.245) we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{rel}(w)\psi[(\text{clr}(X) \text{max } X, \varphi)/X] \quad (C.246)\]

\[\therefore \text{ Case holds by (C.246).} \]

**Case tBlk:** From our rule premises we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{blk}(w)\psi \quad (C.247)\]

because
\[\Gamma = (\Gamma''', w : \text{lid}) \quad (C.248)\]
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma''', w : \text{lid} \vdash \psi \quad (C.249)\]

and
\[\Sigma; \Gamma \vdash \text{max } X, \varphi \quad (C.250)\]

By substitution we know
\[\text{blk}(w)\psi[(\text{clr}(X) \text{max } X, \varphi)/X] \equiv \text{blk}(w)(\psi[(\text{clr}(X) \text{max } X, \varphi)/X]) \quad (C.251)\]

By (C.249), (C.250) and IH we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma'''', w : \text{lid} \vdash \psi[(\text{clr}(X) \text{max } X, \varphi)/X] \quad (C.252)\]

By (C.248), (C.252) and tBlk we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{blk}(w)\psi[(\text{clr}(X) \text{max } X, \varphi)/X] \quad (C.253)\]

By (C.251) and (C.253) we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{blk}(w)\psi[(\text{clr}(X) \text{max } X, \varphi)/X] \quad (C.254)\]

\[\therefore \text{ Case holds by (C.254).} \]

\[\square\]

**To Prove Lemma C.2.4. (Weakening)**

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \varphi \text{ and } X \notin \text{fv}(\varphi) \text{ implies } \Sigma; \Gamma \vdash \varphi.\]

**Proof.** By rule induction on \((\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \varphi.\)
C. Proofs for Auxiliary Lemmas

**Case τNcA:** From our rule premises we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash [e]_a^\alpha \varphi\] (C.255)
because
\[(\Sigma, \{X \mapsto \Gamma\}); (\Gamma, \text{bnd}(e)) \vdash \varphi\] (C.256)
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{rel}(r) \text{tt}\] (C.257)
and
\[X \notin \text{fv}([e]_a^\alpha \varphi)\] (C.258)
From (C.258) we can deduce
\[X \notin \text{fv}(\varphi)\] (C.259)
Since formula variable \(X\) does not feature in formula \(\text{rel}(r) \text{tt}\) we can therefore deduce
\[X \notin \text{fv}(\text{rel}(r) \text{tt})\] (C.260)
By (C.256), (C.259) and IH we know
\[\Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi\] (C.261)
By (C.257), (C.260) and IH we know
\[\Sigma; \Gamma \vdash \text{rel}(r) \text{tt}\] (C.262)
By (C.261), (C.262) and τNcA we know
\[\Sigma; \Gamma \vdash [e]_a^\alpha \varphi\] (C.263)
∴ Case holds by (C.263).

**Case τNcB:** The proof for this case is analogous to that of case τNcA.

**Case τFls:** From our rule premises we know
\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{ff}\] (C.264)
\[X \notin \text{fv}(\text{ff})\] (C.265)
Since (C.264) holds (typechecks) regardless of the contents of the formula environment \(\Sigma\) we can conclude \(\Sigma; \Gamma \vdash \text{ff}\) as required.

**Case τTru:** The proof for this case is analogous to that of case τFls.
C. Proofs for Auxiliary Lemmas

**Case tIf:** From our rule premises we know

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{if } b \text{ then } \varphi \text{ else } \psi\]  \hspace{1cm} \text{(C.266)}

because

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \varphi\]  \hspace{1cm} \text{(C.267)}

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \psi\]  \hspace{1cm} \text{(C.268)}

and

\[X \notin \text{fv}(\text{if } b \text{ then } \varphi \text{ else } \psi)\]  \hspace{1cm} \text{(C.269)}

From (C.269) we can deduce

\[X \notin \text{fv}(\varphi)\]  \hspace{1cm} \text{(C.270)}

\[X \notin \text{fv}(\psi)\]  \hspace{1cm} \text{(C.271)}

By (C.267), (C.270) and IH we know

\[\Sigma; \Gamma \vdash \varphi\]  \hspace{1cm} \text{(C.272)}

By (C.268), (C.271) and IH we know

\[\Sigma; \Gamma \vdash \psi\]  \hspace{1cm} \text{(C.273)}

By (C.272), (C.273) and tIf we know

\[\Sigma; \Gamma \vdash \text{if } b \text{ then } \varphi \text{ else } \psi\]  \hspace{1cm} \text{(C.274)}

\[\therefore\] Case holds by (C.274).

**Case tCn1 and tCn2:** The proofs for these cases are analogous to that of case tIf.

**Case tBlk:** From our rule premises we know

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash \text{blk}(w) \varphi\]  \hspace{1cm} \text{(C.275)}

because

\[(\Sigma, \{X \mapsto \Gamma\}); (\Gamma''', w : \text{lid}) \vdash \varphi\]  \hspace{1cm} \text{(C.276)}

\[\Gamma = \Gamma''', w : \text{lid}\]  \hspace{1cm} \text{(C.277)}

and

\[X \notin \text{fv}((\text{blk}(w) \varphi))\]  \hspace{1cm} \text{(C.278)}
C. Proofs for Auxiliary Lemmas

From (C.278) we can deduce

\[ X \notin \text{fv}(\varphi) \]  
(C.279)

By (C.276), (C.279) and IH we know

\[ \Sigma; (\Gamma'', w : \text{lb}id)\vdash \varphi \]  
(C.280)

By (C.277), (C.280) and \( \tau_{BLK} \) we know

\[ \Sigma; \Gamma \vdash \text{blk}(w) \varphi \]  
(C.281)

\[ \therefore \] Case holds by (C.281).

**Case** \( \tau_{REL}, \tau_{AD}_{A} \) and \( \tau_{AD}_{S} \): The proofs for these cases are analogous to that of case \( \tau_{BLK} \).

**Case** \( \tau_{MAX} \): From our rule premises we know

\[ (\Sigma, \{X \mapsto \Gamma'\}); \Gamma \vdash \text{max} Y. \varphi \]  
(C.282)

because

\[ (\Sigma, \{X \mapsto \Gamma', Y \mapsto \Gamma\}); \Gamma' \vdash \varphi \]  
(C.283)

and

\[ X \notin \text{fv}(\text{max} Y. \varphi) \]  
(C.284)

From (C.284) we can deduce

\[ X \notin \text{fv}(\varphi) \]  
(C.285)

By (C.283), (C.285) and IH we know

\[ (\Sigma, \{Y \mapsto \Gamma\}); \Gamma \vdash \varphi \]  
(C.286)

By (C.286) and \( \tau_{MAX} \) we know

\[ \Sigma; \Gamma \vdash \text{max} Y. \varphi \]  
(C.287)

\[ \therefore \] Case holds by (C.287).

**Case** \( \tau_{CLR} \): The proofs for this case is analogous to that of case \( \tau_{MAX} \).
C. Proofs for Auxiliary Lemmas

Case tVar: From our rule premises we know

\[(\Sigma, \{X \mapsto \Gamma\}); \Gamma \vdash Y\]  \hspace{1cm} (C.288)

because

\[(\Sigma, \{X \mapsto \Gamma\})(Y) \subseteq \Gamma\]  \hspace{1cm} (C.289)

and

\[X \not\in \text{fv}(Y)\]  \hspace{1cm} (C.290)

We now consider two subcases:

Subcase \(Y \neq X\): Since \(Y \neq X\) we know that (C.289) still holds if we remove entry \(X \mapsto \Gamma\) from the formula environment, and hence we deduce

\[\Sigma(Y) \subseteq \Gamma\]  \hspace{1cm} (C.291)

Hence by (C.291) and tVar we can conclude \(\Sigma; \Gamma \vdash Y\) as required.

Subcase \(Y = X\): Since \(Y = X\) we know that (C.290) cannot ever be true. Hence this case holds by contradiction.  \(\Box\)

To Prove Lemma C.2.5. (Term Substitution)

\[\Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi\text{ and } \text{mtch}(e, \alpha) = \sigma \text{ and } \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset\]

and \(\forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma\) implies \(\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \varphi\sigma\)

Proof. By rule induction on \(\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \varphi\).

Case tFls: From our rule premises we know

\[\Sigma; (\Gamma, \text{bnd}(e)) \vdash \text{ff}\]  \hspace{1cm} (C.292)

\[\text{mtch}(e, \alpha) = \sigma\]  \hspace{1cm} (C.293)

\[\text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset\]  \hspace{1cm} (C.294)

\[\forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma\]  \hspace{1cm} (C.295)

Since \text{ff} has no term variables that can be substituted, we know

\[\text{ff}\sigma \equiv \text{ff}\]  \hspace{1cm} (C.296)

By (C.292), (C.296) and given that \text{ff} typechecks with any type environment \(\Gamma\) we can conclude

\[\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \text{ff}\sigma\]  \hspace{1cm} (C.297)
C. Proofs for Auxiliary Lemmas

\[ \therefore \text{Case holds by (C.297)}. \]

**Note:** The proof for cases \( \tau \text{Tru} \) and \( \tau \text{Var} \) are analogous to the proof for case \( \tau \text{Fls} \).

**Case** \( \tau \text{Max} \): From our rule premises we know

\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \text{max} \cdot X. \varphi \quad (C.298) \]

because

\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi \quad (C.299) \]

and

\[ \text{mtch}(e, \alpha) = \sigma \quad (C.300) \]

\[ \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \quad (C.301) \]

\[ \forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \quad (C.302) \]

By (C.299), (C.300), (C.301), (C.302) and IH we know

\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash (\varphi\sigma) \quad (C.303) \]

By (C.303) and \( \tau \text{Max} \) we know

\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \text{max} \cdot X. (\varphi\sigma) \quad (C.304) \]

By (C.304) and data substitution we know

\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash (\text{max} \cdot X. \varphi)\sigma \quad (C.305) \]

\[ \therefore \text{Case holds by (C.305)}. \]

**Case** \( \tau \text{Clr} \): The proof for this case is analogous to that of case \( \tau \text{Max} \)

**Case** \( \tau \text{AdS} \): From our rule premises we know

\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash sA(w). \varphi \quad (C.306) \]

because

\[ \Gamma, \text{bnd}(e) = \Gamma', w : \text{lid} \quad (C.307) \]

\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \text{rel}(r) \varphi \quad (C.308) \]
C. Proofs for Auxiliary Lemmas

and

\[
\text{mtch}(e, a) = \sigma \tag{C.309}
\]

\[
\text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \tag{C.310}
\]

\[
\forall (x, y) : \text{lid} \in \text{bnd}(e) : x \neq y \implies x\sigma \neq y\sigma \tag{C.311}
\]

By (C.308), (C.309), (C.310), (C.311) and IH we know

\[
\Sigma ; (\Gamma, \text{bnd}(e)\sigma) \vdash (sA(w), (\varphi\sigma)) \tag{C.312}
\]

By (C.310) and (C.311) we know that we do not have any aliasing, hence we can safely apply substitution \(\sigma\) on both sides of equation (C.307), such that we know

\[
\Gamma, \text{bnd}(e)\sigma = (\Gamma', w : \text{lid})\sigma \tag{C.313}
\]

By (C.312), (C.313) and \(\text{tAdS}\) we know

\[
\Sigma ; (\Gamma, \text{bnd}(e)\sigma) \vdash sA(w), (\varphi\sigma) \tag{C.314}
\]

By (C.314) and data substitution we know

\[
\Sigma ; (\Gamma, \text{bnd}(e)\sigma) \vdash (sA(w), \varphi\sigma) \tag{C.315}
\]

\(\therefore\) Case holds by (C.315).

Case \(\text{tAdA}\): The proof for this case is very similar to the proof of cases \(\text{tAdS}\).

Case \(\text{tIf}\): From our rule premises we know

\[
\Sigma ; ((\Gamma, \text{bnd}(e)) + \text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2) \tag{C.316}
\]

because

\[
\Sigma ; (\Gamma, \text{bnd}(e)) + \varphi_1 \tag{C.317}
\]

\[
\Sigma ; (\Gamma, \text{bnd}(e)) + \varphi_2 \tag{C.318}
\]

and

\[
\text{mtch}(e, a) = \sigma \tag{C.319}
\]

\[
\text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \tag{C.320}
\]

\[
\forall (x, y) : \text{lid} \in \text{bnd}(e) : x \neq y \implies x\sigma \neq y\sigma \tag{C.321}
\]

By (C.317), (C.319), (C.320), (C.321) and IH we know

\[
\Sigma ; (\Gamma, \text{bnd}(e)\sigma) \vdash \varphi_1\sigma \tag{C.322}
\]
By (C.318), (C.319), (C.320), (C.321) and IH we know
\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \varphi_2\sigma \] (C.323)

By (C.322), (C.323) and \texttt{tI} we know
\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \text{if } c\sigma \text{ then } (\varphi_1\sigma) \text{ else } (\varphi_2\sigma) \] (C.324)

By (C.324) and data substitution we know
\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash (\text{if } c\sigma \text{ then } \varphi_1\sigma \text{ else } \varphi_2\sigma) \] (C.325)

\[ \therefore \text{ Case holds by (C.325).} \]

\textbf{Case } \texttt{tCn1:} From our rule premises we know
\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi_1 \& \varphi_2 \] (C.326)

because
\[ \Gamma, \text{bnd}(e) = (\Gamma_1 + \Gamma_2) \] (C.327)
\[ \Sigma; \Gamma_1 \vdash \varphi_1 \] (C.328)
\[ \Sigma; \Gamma_2 \vdash \varphi_2 \] (C.329)
\[ \text{excl}(\varphi, \psi) = \bot \] (C.330)

and
\[ \text{mtch}(e, \alpha) = \sigma \] (C.331)
\[ \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \] (C.332)
\[ \forall (x, y): \text{lid} \in \text{bnd}(e): x \neq y \implies x\sigma \neq y\sigma \] (C.333)

By (C.328), (C.331), (C.332), (C.333) and IH we know
\[ \Sigma; (\Gamma_1\sigma) \vdash \varphi_1\sigma \] (C.334)

By (C.329), (C.331), (C.332), (C.333) and IH we know
\[ \Sigma; (\Gamma_2\sigma) \vdash \varphi_2\sigma \] (C.335)

By (C.332) and (C.333) we know that we do not have any aliasing, hence we can safely apply substitution \( \sigma \) on both sides of equation (C.327), such that we know
\[ \Gamma, \text{bnd}(e)\sigma = (\Gamma_1 + \Gamma_2)\sigma \] (C.336)

By (C.334), (C.335), (C.336) and Lemma [C.3.1] we know
\[ \Sigma; (\Gamma_1 + \Gamma_2)\sigma \vdash (\varphi_1\sigma) \& (\varphi_2\sigma) \] (C.337)
By (C.327), (C.337) and data substitution we know
\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash (\varphi_1 \land \varphi_2) \sigma \]  \hspace{1cm} (C.338)

\[ \therefore \text{Case holds by (C.338).} \]

**Case \( \tau \text{Cn}2: \)** From our rule premises we know
\[ \Sigma; (\Gamma, \text{bnd}(e)) \vdash \varphi_1 \land \varphi_2 \]  \hspace{1cm} (C.339)
because
\[ \text{excl}(\psi_1, \psi_2) = (r_1, r_2) \]  \hspace{1cm} (C.340)
\[ \Sigma; \text{eff}(\Gamma, \text{bnd}(e)), r_2 \vdash \varphi_1 \]  \hspace{1cm} (C.341)
\[ \Sigma; \text{eff}(\Gamma, \text{bnd}(e)), r_1 \vdash \varphi_2 \]  \hspace{1cm} (C.342)

and
\[ \text{mtch}(e, \alpha) = \sigma \]  \hspace{1cm} (C.343)
\[ \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e) \sigma) = \emptyset \]  \hspace{1cm} (C.344)
\[ \forall (x, y): \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \]  \hspace{1cm} (C.345)

As by (C.340) we know that patterns \( \psi_1 \) and \( \psi_2 \) are exclusive, if we add more information by applying substitution (C.343), we do not affect their mutual exclusion. Moreover, the substitution can safely be applied as by (C.344) and (C.345) we know that we do not have aliasing. Hence we can deduce
\[ \text{excl}(\varphi_1 \sigma, \varphi_2 \sigma) = (r_1 \sigma, r_2 \sigma) \]  \hspace{1cm} (C.346)

By (C.341), (C.343), (C.344), (C.345) and IH we know
\[ \Sigma; \text{eff}(\Gamma, \text{bnd}(e)), r_2 \sigma \vdash \varphi_1 \]  \hspace{1cm} (C.347)

By (C.342), (C.343), (C.344), (C.345) and IH we know
\[ \Sigma; \text{eff}(\Gamma, \text{bnd}(e)), r_1 \sigma \vdash \varphi_2 \]  \hspace{1cm} (C.348)

By (C.346), (C.347), (C.348) and \( \tau \text{Cn}2 \) we know
\[ \Sigma; (\Gamma, \text{bnd}(e) \sigma) \vdash (\varphi_1 \sigma) \land (\varphi_2 \sigma) \]  \hspace{1cm} (C.349)

By (C.349) and substitution we know
\[ \Sigma; (\Gamma, \text{bnd}(e) \sigma) \vdash (\varphi_1 \land \varphi_2) \sigma \]  \hspace{1cm} (C.350)

\[ \therefore \text{Case holds by (C.350).} \]
C. Proofs for Auxiliary Lemmas

**Case tNcB:** From our rule premises we know

\[
\Sigma; (\Gamma, \text{bnd}(e)) \vdash [d]^b \varphi \tag{C.351}
\]

because

\[
\text{subj}(d) = l \tag{C.352}
\]

\[
\Sigma; ((\Gamma, \text{bnd}(e)), \text{bnd}(d)) \vdash \text{blk}(l) \varphi \tag{C.353}
\]

\[
\Sigma; (\Gamma, \text{bnd}(e)) \vdash \text{rel}(r) \text{tt} \tag{C.354}
\]

and

\[
\text{mtch}(e, \alpha) = \sigma \tag{C.355}
\]

\[
\text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\alpha) = \emptyset \tag{C.356}
\]

\[
\forall (x, y): \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \tag{C.357}
\]

By (C.356) and (C.357) we know that we do not have any aliasing, hence we can safely apply substitution \(\sigma\) on both sides of equation (C.352), such that we know

\[
\text{subj}(d\sigma) = l\sigma \tag{C.358}
\]

By (C.353), (C.355), (C.356), (C.357) and IH we know

\[
\Sigma; ((\Gamma, \text{bnd}(e)), \text{bnd}(d))\sigma \vdash (\text{blk}(l\sigma) (\varphi\sigma)) \tag{C.359}
\]

By (C.354), (C.355), (C.356), (C.357) and IH we know

\[
\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \text{rel}(r\sigma) \text{tt} \tag{C.360}
\]

By (C.358), (C.359), (C.360) and tNcB we know

\[
\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash [d]^b \sigma (\varphi\sigma) \tag{C.361}
\]

By (C.361) and data substitution we know

\[
\Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash ([d]^b \varphi)\sigma \tag{C.362}
\]

\[
\therefore \text{Case holds by (C.362).}
\]

**Case tNcA:** The proof for this case is analogous to that of case tNcB.

**Case tBlk:** From our rule premises we know

\[
\Sigma; (\Gamma, \text{bnd}(e)) \vdash \text{blk}(w) \varphi \tag{C.363}
\]
because
\[ \Gamma, \text{bnd}(e) = \Gamma', w : \text{lid} \quad \text{(C.364)} \]
\[ \Sigma; (\Gamma', w : \text{lid}) \vdash \varphi \quad \text{(C.365)} \]

and
\[ \text{mtch}(e, \alpha) = \sigma \quad \text{(C.366)} \]
\[ \text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \quad \text{(C.367)} \]
\[ \forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \quad \text{(C.368)} \]

By (C.365), (C.366), (C.367), (C.368) and IH we know
\[ \Sigma; (\Gamma', w : \text{lid})\sigma \vdash \varphi\sigma \quad \text{(C.369)} \]

By (C.367) and (C.368) we know that we do not have any aliasing, hence we can safely apply substitution \( \sigma \) on both sides of equation (C.364), such that we know
\[ \Gamma, \text{bnd}(e)\sigma = (\Gamma', w : \text{lid})\sigma \quad \text{(C.370)} \]

By (C.369), (C.370) and \( \text{tBlk} \) we know
\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash \text{blk}(w) (\varphi\sigma) \quad \text{(C.371)} \]

By (C.371) and data substitution we know
\[ \Sigma; (\Gamma, \text{bnd}(e)\sigma) \vdash (\text{blk}(w) \varphi)\sigma \quad \text{(C.372)} \]

\( \therefore \) Case holds by (C.372).

Case \( \text{tRel} \): The proof for this case is analogous to that of case \( \text{tBlk} \). \( \square \)

\section*{C.3 Proving The Secondary Soundness Auxiliary Lemmas}

\emph{To Prove Lemma C.3.1.} \( \Sigma; \Gamma_1 \vdash \varphi \) and \( \Sigma; \Gamma_2 \vdash \psi \) and \( \Gamma = \Gamma_1 + \Gamma_2 \) implies \( \Sigma; \Gamma \vdash \varphi \& \psi \).

\emph{Proof.} From our premises we know
\[ \Sigma; \Gamma_1 \vdash \varphi \quad \text{(C.373)} \]
\[ \Sigma; \Gamma_2 \vdash \psi \quad \text{(C.374)} \]
\[ \Gamma = \Gamma_1 + \Gamma_2 \quad \text{(C.375)} \]
We consider two cases:

**Case** \( \text{excl}(\varphi, \psi) = \bot \): By (C.373), (C.374), (C.375) and \( \tau \text{Cn1} \) we know

\[
\Sigma; \Gamma \vdash \varphi \land \psi \quad \text{(C.376)}
\]

∴ Case holds by (C.376).

**Case** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \): By (C.373) and since \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) we can deduce that since \( r_\varphi \) represents a set of processes that may be released by \( \varphi \) in case this is trivially satisfied, then we know that these processes are linear typed. Hence since by (C.375) we know that \( \Gamma_1 \) and \( \Gamma_2 \) do not share linear typed references we can conclude

\[
\forall l \in r_\varphi \cdot l : \text{lin} \in \Gamma_1 \land l : \text{lin} \not\in \Gamma_2 \quad \text{(where lin = lid or lbid)} \quad \text{(C.377)}
\]

Same argument applies for (C.374), such that we know

\[
\forall l \in r_\psi \cdot l : \text{lin} \in \Gamma_2 \land l : \text{lin} \not\in \Gamma_1 \quad \text{(C.378)}
\]

Hence by (C.373), (C.375), (C.378) and defn of \( \text{eff} \) (Def. 5.1.1) we know that the side-effects of \( \psi \) (i.e., when the processes in \( r_\psi \) are released) do not interfere with \( \varphi \), as these branches do not share linear variables, such that we can deduce

\[
\Sigma; \text{eff}(\Gamma_1 + \Gamma_2, r_\varphi) \vdash \varphi \equiv \Sigma; \text{eff}(\Gamma_1, r_\varphi) \vdash \varphi \quad \text{(C.379)}
\]

Similarly by (C.374), (C.375), (C.377) and defn of \( \text{eff} \) (Def. 5.1.1) we can deduce

\[
\Sigma; \text{eff}(\Gamma_1 + \Gamma_2, r_\psi) \vdash \psi \equiv \Sigma; \text{eff}(\Gamma_1, r_\psi) \vdash \psi \quad \text{(C.380)}
\]

Since \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \), by (C.379), (C.380) and \( \tau \text{Cn2} \) we know

\[
\Sigma; \Gamma \vdash \varphi \land \psi \quad \text{(C.381)}
\]

∴ Case holds by (C.381).

**To Prove Lemma C.3.2.** \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \) and \( \Sigma; \text{eff}(\Gamma, r) \vdash \varphi \) implies \( \langle \text{eff}(\Gamma, r), \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \text{eff}(\text{after}(\Gamma, \gamma), r), \Delta', \varphi' \rangle \) and \( \Gamma' = \text{after}(\Gamma, \gamma) \)

**Proof.** By rule induction on \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \).

**Case** \( \text{rTru} \): From our rule premises we know

\[
\langle \Gamma, \Delta, \text{if } b \text{ then } \varphi \text{ else } \psi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta, \varphi \rangle \quad \text{(C.382)}
\]

\[
\Sigma; \text{eff}(\Gamma, r) \vdash \text{if } b \text{ then } \varphi \text{ else } \psi \quad \text{(C.383)}
\]

Since \( \gamma = \tau \) and by defn of \( \text{after} \) (Def. 5.3.2) we know

\[
\Gamma' = \text{after}(\Gamma, \tau) \quad \text{(C.384)}
\]
C. Proofs for Auxiliary Lemmas

By (C.383) and \( \tau \text{If} \) we know
\[
\Sigma; \text{eff}(\Gamma, r) \vdash \varphi
\]  
(C.385)

Since reduction (C.382) can be performed regardless of the contents of \( \Gamma \) and by (C.383), (C.384) and (C.385) we can deduce
\[
\langle \text{eff}(\Gamma, r), \Delta, \text{if } b \text{ then } \varphi \text{ else } \psi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, r)), \Delta, \varphi \rangle
\]  
(C.386)

\( \therefore \) Case holds by (C.384) and (C.386).

**Note:** The proofs for cases \( rFls, rClr \) and \( rMax \) are analogous to the proof of case \( rTru \).

**Case** \( rRel \): From our rule premises we know
\[
\langle (\Gamma, w : \text{lid}), \Delta, \text{rel}(w) \varphi \rangle \xrightarrow{r(w)} \langle (\Gamma, w : \text{lid}), \Delta, \varphi \rangle
\]  
(C.387)
\[
\Sigma; \text{eff}(\langle (\Gamma, w : \text{lid}), r(w) \rangle) \vdash \varphi
\]  
(C.388)

Since \( \gamma = r(w) \) and by defn of after (Def. 5.3.2) we know
\[
(\Gamma, w : \text{lid}) = \text{after}(\langle (\Gamma, w : \text{lid}), r(w) \rangle)
\]  
(C.389)

By (C.388) and \( \tau \text{Rel} \) we know
\[
\Sigma; \text{eff}(\langle (\Gamma, w : \text{lid}), r \rangle) \vdash \varphi
\]  
(C.390)

By (C.388) we know that even if we apply the \( \text{eff} \) function on environment \( (\Gamma, w : \text{lid}) \), reduction (C.387) can still be performed. Hence by (C.390) and the defn of after (Def. 5.3.2) we can deduce
\[
\langle \text{eff}(\langle (\Gamma, w : \text{lid}), r(w) \rangle), \Delta, \varphi \rangle \xrightarrow{r(w)} \langle \text{eff}(\text{after}(\Gamma, w : \text{lid}), r(w)), r, \Delta, \varphi \rangle
\]  
(C.391)

\( \therefore \) Case holds by (C.389) and (C.391).

**Note:** The proofs for cases \( rBlk, rAda \) and \( rAdS \) are analogous to the proof of case \( rRel \).

**Case** \( rNc1 \): From our rule premises we know
\[
(\Gamma, \Delta, \lbrack e_i \rbrack^b_w \varphi) \xrightarrow{\alpha} (\Gamma', \Gamma', \text{blk}(i) \varphi \sigma)
\]  
(C.392)
because

\[
\text{mtch}(e, \alpha) = \sigma \quad \text{(C.393)}
\]
\[
\text{subj}(\alpha) = i \quad \text{(C.394)}
\]
\[
\Gamma' = \text{bnd}(e)\sigma \quad \text{(C.395)}
\]
\[
dom(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) = \emptyset \quad \text{(C.396)}
\]
\[
\forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \quad \text{(C.397)}
\]

and

\[
\Sigma; \text{eff}(\Gamma, r) \vdash [e]_\alpha^\sigma \varphi \quad \text{(C.398)}
\]

Since \( \gamma = \alpha \), \( \Gamma \subseteq (\Gamma, \Gamma') \) and by defn of after (Def. 5.3.2) we know

\[
(\Gamma, \Gamma') = \text{after}(\Gamma, \alpha) \quad \text{(C.399)}
\]

By (C.398) and \( \tau\text{NcB} \) we know

\[
\text{subj}(e) = l \quad \text{(C.400)}
\]
\[
\Sigma; (\text{eff}(\Gamma, r), \text{bnd}(e)) \vdash \text{blk}(l) \varphi \quad \text{(C.401)}
\]

Since newly bound variables cannot be of the type \( \text{lid} \), then by the defn of \( \text{eff} \) (Def. 5.1.1) we know

\[
\text{eff}(\text{bnd}(e), r) \equiv \text{bnd}(e) \quad \text{(C.402)}
\]

Hence by (C.401) and (C.402) we know

\[
\Sigma; (\text{eff}((\Gamma, \text{bnd}(e)), r)) \vdash \text{blk}(l) \varphi \quad \text{(C.403)}
\]

By (C.394) and (C.400) we know that \( l\sigma = i \), hence by (C.393), (C.396), (C.397), (C.403) and Lemma C.2.5 (Term Substitution) we know

\[
\Sigma; (\text{eff}((\Gamma, \text{bnd}(e)\sigma), r)) \vdash \text{blk}(i) \varphi\sigma \quad \text{(C.404)}
\]

By (C.398) and (C.403) we know that even if we apply the \( \text{eff} \) function on environment \( \Gamma \), reduction (C.392) can still be performed, as by (C.404) we know that it reduces into a form which also typechecks. Hence by (C.404) and the defn of after (Def. 5.3.2) we can deduce

\[
\langle \text{eff}(\Gamma, r), \Delta, [e]_\alpha^\sigma \varphi \rangle \xrightarrow{\alpha} \langle \text{eff}(\text{after}(\Gamma, \alpha), r), \Delta, \text{blk}(i) \varphi\sigma \rangle \quad \text{(C.405)}
\]

\[ \therefore \text{Case holds by (C.399) and (C.405).} \]

**Case rNc2**: The proof for this case is analogous to the proof for case rNc1.
C. Proofs for Auxiliary Lemmas

**Case rNc3**: From our rule premises we know
\[
\langle \Gamma, \Delta, [e]_w^\varphi \rangle \xrightarrow{\alpha} \langle \Gamma, \Delta, \text{rel}(w) \text{tt} \rangle
\]  
(C.406)

because
\[
\text{mtch}(e, \alpha) = \bot
\]  
(C.407)

and
\[
\Sigma; \text{eff}(\Gamma, r) \vdash [e]_w^\varphi
\]  
(C.408)

Since \( \gamma = \alpha \), \( \Gamma \subseteq \Gamma \) and by defn of \textit{after} (Def. 5.3.2) we know
\[
\Gamma = \text{after}(\Gamma, \alpha)
\]  
(C.409)

Since \( \rho \in \{b, a\} \) we must consider two subcases:

**Subcase \( \rho = b \)**: By (C.408) and rNcB we know
\[
\Sigma; \text{eff}(\Gamma, r) \vdash \text{rel}(w) \text{tt}
\]  
(C.410)

By (C.408) and (C.410) we know that reduction (C.406) can still be performed regardless of the contents of \( \Gamma \), hence by (C.409) we can deduce
\[
\langle \text{eff}(\Gamma, r), \Delta, [e]_w^\varphi \rangle \xrightarrow{\alpha} \langle \text{eff}(\Gamma, r), \Delta, \text{rel}(w) \text{tt} \rangle
\]  
(C.411)

\(\because\) Subcase holds by (C.409) and (C.411).

**Subcase \( \rho = a \)**: The proof for this subcase is analogous to that of the previous subcase (i.e., \( \rho = b \)).

**Case rCn1**: From our rule premises we know
\[
\langle \Gamma, \Delta, \varphi \& \psi \rangle \xrightarrow{\alpha} \langle (\Gamma', \Gamma''), \Delta' \cup \Delta'', \varphi' \& \psi' \rangle
\]  
(C.412)

because
\[
\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle
\]  
(C.413)
\[
\langle \Gamma, \Delta, \psi \rangle \xrightarrow{\alpha} \langle \Gamma'', \Delta'', \psi' \rangle
\]  
(C.414)

\[\text{dom}(\Delta) = \text{dom}(\Delta') \cap \text{dom}(\Delta'')\]  
(C.415)

and
\[
\Sigma; \text{eff}(\Delta, r) \vdash \varphi \& \psi
\]  
(C.416)

We must consider two subcases:
Subcase excl(ϕ, ψ) = (rϕ, rψ): Since excl(ϕ, ψ) = (rϕ, rψ), by (C.416) and rCn2 we know
\[ \Sigma; \text{eff}(\Gamma, rϕ) \vdash ϕ \]  
\[ \Sigma; \text{eff}(\Gamma, rψ) \vdash ψ \]  
(C.417)  
(C.418)

By (C.413), (C.417) and IH we know
\[ \langle \text{eff}(\Gamma, rrϕ), Δ, ϕ \rangle \xrightarrow{α} \langle \text{eff}(\text{after}(Γ, α), rrϕ), Δ', ϕ' \rangle \]  
\[ Γ' = \text{after}(Γ, α) \]  
(C.419)  
(C.420)

By (C.414), (C.418) and IH we know
\[ \langle \text{eff}(\Gamma, rrψ), Δ, ψ \rangle \xrightarrow{α} \langle \text{eff}(\text{after}(Γ, α), rrψ), Δ'', ψ' \rangle \]  
\[ Γ'' = \text{after}(Γ, α) \]  
(C.421)  
(C.422)

By (C.413) and (C.419) we know that formula ϕ can reduce into ϕ' regardless of whether we apply the side-effects or not. Hence we can deduce
\[ \langle \text{eff}(Γ, r), Δ, ϕ \rangle \xrightarrow{α} \langle \text{eff}(\text{after}(Γ, α), r), Δ', ϕ' \rangle \]  
(C.423)

The same argument applies by (C.414) and (C.421), such that we know we know
\[ \langle \text{eff}(Γ, r), Δ, ψ \rangle \xrightarrow{α} \langle \text{eff}(\text{after}(Γ, α), r), Δ'', ψ' \rangle \]  
(C.424)

By (C.420), (C.422) and the defn of after (Def. 5.3.2) we know
\[ Γ \subseteq Γ' \]  
\[ Γ \subseteq Γ'' \]  
(C.425)  
(C.426)

By (C.425) and (C.426) we know
\[ Γ \subseteq (Γ', Γ'') \]  
(C.427)

By (C.427) and and the defn of after (Def. 5.3.2) we know
\[ (Γ', Γ'') = \text{after}(Γ, α) \]  
(C.428)

By (C.415), (C.423), (C.424), (C.428) and rCn1 we know
\[ \langle \text{eff}(Γ, r), Δ, ϕ & ψ \rangle \xrightarrow{α} \langle \text{eff}(\text{after}(Γ, α), r), Δ' \cup Δ'', ϕ' & ψ' \rangle \]  
(C.429)

∴ Subcase holds by (C.428) and (C.429).

Subcase excl(ϕ, ψ) = ⊥: Since excl(ϕ, ψ) = ⊥, by (C.416) and rCn1 we know
\[ \Sigma; \text{eff}(Γ_1, r) \vdash ϕ \]  
\[ \Sigma; \text{eff}(Γ_2, r) \vdash ψ \]  
\[ \text{eff}(Γ, r) = \text{eff}(Γ_1, r) + \text{eff}(Γ_2, r) \]  
(C.430)  
(C.431)  
(C.432)
Since by (C.432) we know \( \text{eff}(\Gamma, r) \subseteq \text{eff}(\Gamma, r) \) and \( \text{eff}(\Gamma_2, r) \subseteq \text{eff}(\Gamma, r) \), then by (C.430) and (C.431) we can deduce
\[
\Sigma; \text{eff}(\Gamma, r) \vdash \varphi \tag{C.433}
\]
\[
\Sigma; \text{eff}(\Gamma, r) \vdash \psi \tag{C.434}
\]
By (C.413), (C.433) and IH we know
\[
(\text{eff}(\Gamma, r), \Delta, \varphi) \overset{\alpha}{\rightarrow} (\text{eff}(\text{after}(\Gamma, \alpha), r), \Delta', \varphi') \tag{C.435}
\]
\[
\Gamma' = \text{after}(\Gamma, \alpha) \tag{C.436}
\]
By (C.414), (C.434) and IH we know
\[
(\text{eff}(\Gamma, r), \Delta, \varphi) \overset{\alpha}{\rightarrow} (\text{eff}(\text{after}(\Gamma, \alpha), r), \Delta'', \psi') \tag{C.437}
\]
\[
\Gamma'' = \text{after}(\Gamma, \alpha) \tag{C.438}
\]
By (C.436), (C.438) and the defn of after (Def. 5.3.2) we know
\[
(\Gamma', \Gamma'') = \text{after}(\Gamma, \alpha) \tag{C.439}
\]
By (C.415), (C.435), (C.437), (C.439) and rCn1 we know
\[
(\text{eff}(\Gamma, r), \Delta, \varphi \& \psi) \overset{\alpha}{\rightarrow} (\text{eff}(\text{after}(\Gamma, \alpha), r), \Delta' \cup \Delta'', \varphi' \& \psi') \tag{C.440}
\]
\[
 \therefore \text{Subcase holds by (C.439) and (C.440)}.
\]
**Case rCn2:** From our rule premises we know
\[
(\Gamma, \Delta, \varphi \& \psi) \overset{\tau}{\rightarrow} (\Gamma, \Delta', \varphi' \& \psi) \tag{C.441}
\]
because
\[
(\Gamma, \Delta, \varphi) \overset{\tau}{\rightarrow} (\Gamma', \Delta', \varphi') \tag{C.442}
\]
and
\[
\Sigma; \text{eff}(\Gamma, r) \vdash \varphi \& \psi \tag{C.443}
\]
We must consider two subcases:

**Subcase excl(\( \varphi, \psi \)) = (r_{\varphi}, r_{\psi})**: Since excl(\( \varphi, \psi \)) = (r_{\varphi}, r_{\psi}), by (C.443) and rCn2 we know
\[
\Sigma; \text{eff}(\Gamma, r_{\varphi}) \vdash \varphi \tag{C.444}
\]
\[
\Sigma; \text{eff}(\Gamma, r_{\psi}) \vdash \psi \tag{C.445}
\]

\[
200
\]
By (C.442), (C.444) and IH we know
\[ \langle \text{eff}(\Gamma, rr\varphi), \Delta, \varphi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, \tau), rr\varphi), \Delta', \varphi' \rangle \] (C.446)
\[ \Gamma = \text{after}(\Gamma, \tau) \] (C.447)
By (C.442) and (C.446) we know that formula \( \varphi \) can reduce into \( \varphi' \) regardless of whether we apply the side-effects or not. Hence we can deduce
\[ \langle \text{eff}(\Gamma, r), \Delta, \varphi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, \tau), r), \Delta', \varphi' \rangle \] (C.448)
By (C.448) and \( \text{rCn2} \) we know
\[ \langle \text{eff}(\Gamma, r), \Delta, \varphi \& \psi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, \tau), r), \Delta', \varphi' \& \psi \rangle \] (C.449)
\[ \therefore \text{Subcase holds by (C.447) and (C.449)}. \]

**Subcase** \( \text{excl}(\varphi, \psi) = \bot \): Since \( \text{excl}(\varphi, \psi) = \bot \), by (C.443) and \( \text{rCn1} \) we know
\[ \Sigma; \text{eff}(\Gamma_1, r) + \varphi \] (C.450)
\[ \Sigma; \text{eff}(\Gamma_2, r) + \psi \] (C.451)
\[ \text{eff}(\Gamma, r) = \text{eff}(\Gamma_1, r) + \text{eff}(\Gamma_2, r) \] (C.452)
Since by (C.452) we know \( \text{eff}(\Gamma_1, r) \subseteq \text{eff}(\Gamma, r) \) and \( \text{eff}(\Gamma_2, r) \subseteq \text{eff}(\Gamma, r) \), then by (C.450) we can deduce
\[ \Sigma; \text{eff}(\Gamma, r) + \varphi \] (C.453)
By (C.442), (C.453) and IH we know
\[ \langle \text{eff}(\Gamma, r), \Delta, \varphi \& \psi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, \tau), r), \Delta', \varphi' \& \psi \rangle \] (C.454)
\[ \Gamma = \text{after}(\Gamma, \tau) \] (C.455)
By (C.454) and \( \text{rCn2} \) we know
\[ \langle \text{eff}(\Gamma, r), \Delta, \varphi \& \psi \rangle \xrightarrow{\tau} \langle \text{eff}(\text{after}(\Gamma, \tau), r), \Delta', \varphi' \& \psi \rangle \] (C.456)
\[ \therefore \text{Subcase holds by (C.455) and (C.456)}. \]

**Case** \( \text{rCn3} \): The proof for this case is analogous to the proof of case \( \text{rCn2} \). \( \square \)

**To Prove Lemma C.3.3.** \( \Gamma = \Gamma_1 + \Gamma_2 \) and \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \) and \( \Sigma; \Gamma_1 + \varphi \)

implies \( \langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma_1', \Delta', \varphi' \rangle \) and \( \Gamma' = \Gamma_1' + \Gamma_2 \)

**Proof.** By rule induction on \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\gamma} \langle \Gamma', \Delta', \varphi' \rangle \).
C. Proofs for Auxiliary Lemmas

**Case rTru:** From our rule premises we know

\[ \langle \Gamma, \Delta, \text{if } b \text{ then } \varphi \text{ else } \psi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta, \varphi \rangle \quad (C.457) \]

\[ \Sigma; \Gamma_1 \vdash \text{if } b \text{ then } \varphi \text{ else } \psi \quad (C.458) \]

\[ \Gamma = \Gamma_1 + \Gamma_2 \quad (C.459) \]

By (C.458) and \( \tau \text{If} \) we know

\[ \Sigma; \Gamma_1 \vdash \varphi \quad (C.460) \]

Since reduction (C.457) can be performed regardless of the contents of \( \Gamma \) and by (C.458) and (C.460) we can deduce

\[ \langle \Gamma_1, \Delta, \text{if } b \text{ then } \varphi \text{ else } \psi \rangle \xrightarrow{\tau} \langle \Gamma_1, \Delta, \varphi \rangle \quad (C.461) \]

\[ \therefore \text{ Case holds by (C.459) and (C.461).} \]

**Note:** The proofs for cases rFls, rClr and rMax are analogous to the proof of case rTru.

**Case rRel:** From our rule premises we know

\[ \langle (\Gamma, w: \text{bid}), \Delta, \text{rel}(w) \varphi \rangle \xrightarrow{\tau} \langle (\Gamma, w: \text{lid}), \Delta, \varphi \rangle \quad (C.462) \]

\[ \Sigma; \Gamma_1 \vdash \text{rel}(w) \varphi \quad (C.463) \]

\[ (\Gamma, w: \text{lid}) = \Gamma_1 + \Gamma_2 \quad (C.464) \]

By (C.463) and \( \tau \text{Rel} \) we know

\[ \Gamma_1 = (\Gamma'_1, w: \text{bid}) \quad (C.465) \]

\[ \Gamma'_1 = (\Gamma''_1, w: \text{lid}) \quad (C.466) \]

\[ \Sigma; \Gamma'_1 \vdash \varphi \quad (C.467) \]

By (C.465) and (C.466) we know that reduction (C.462) can still be performed using subset \( \Gamma_1 \) instead of \( \Gamma \). Hence we can deduce

\[ \langle \Gamma_1, \Delta, \text{rel}(w) \varphi \rangle \xrightarrow{\tau} \langle \Gamma'_1, \Delta, \varphi \rangle \quad (C.468) \]

By (C.464) and (C.466) we can conclude

\[ (\Gamma, w: \text{lid}) = \Gamma'_1 + \Gamma_2 \quad (C.469) \]

\[ \therefore \text{ Case holds by (C.468) and (C.469).} \]
C. Proofs for Auxiliary Lemmas

Note: The proofs for cases \(r\)\(\text{BLK}\), \(r\)\(\text{ADA}\) and \(r\)\(\text{ADS}\) are analogous to the proof of case \(r\)\(\text{REL}\).

Case \(r\)\(\text{NC1}\): From our rule premises we know
\[
\langle \Gamma, \Delta, [e]_r^b \varphi \rangle \xrightarrow{\alpha} \langle (\Gamma, \Gamma'), \Delta', \text{blk}(i) \varphi \sigma \rangle \quad (C.470)
\]
because
\[
\begin{align*}
\text{mtch}(e, \alpha) &= \sigma \quad (C.471) \\
\text{subj}(\alpha) &= i \quad (C.472) \\
\Gamma' &= \text{bnd}(e)\sigma \quad (C.473) \\
\text{dom}(\Delta) \cap \text{dom}(\text{bnd}(e)\sigma) &= \emptyset \quad (C.474) \\
\forall (x, y) : \text{lid} \in \text{bnd}(e) \cdot x \neq y \implies x\sigma \neq y\sigma \quad (C.475)
\end{align*}
\]
and
\[
\begin{align*}
\Sigma; \Gamma_1 + [e]_r^b \varphi & \\
\Gamma &= \Gamma_1 + \Gamma_2
\end{align*}
\]
By (C.476) and \(\tau\)\(\text{NCB}\) we know
\[
\Sigma; (\Gamma_1, \text{bnd}(e)) + \text{blk}(l) \varphi \quad (C.478)
\]
By (C.471), (C.474), (C.475), (C.478) and Lemma\(C.2.5\) (Term Substitution) we know
\[
\Sigma; (\Gamma_1, \text{bnd}(e)\sigma) + \text{blk}(i) \varphi \sigma \quad (C.479)
\]
Since by (C.473) and (C.479) we know that \(\text{blk}(i) \varphi \sigma\) typechecks with \((\Gamma_1, \Gamma')\), this means that reduction (C.470) can still be performed using subset \(\Gamma_1\) instead of \(\Gamma\), such that we know
\[
\langle \Gamma_1, \Delta, [e]_r^b \varphi \rangle \xrightarrow{\alpha} \langle (\Gamma_1, \Gamma'), \Delta', \text{blk}(i) \varphi \sigma \rangle \quad (C.480)
\]
By (C.477) and (C.473) we can conclude
\[
(\Gamma, \Gamma') = (\Gamma_1, \Gamma') + \Gamma_2 \quad (C.481)
\]
\(\therefore\) Case holds by (C.480) and (C.481).

Case \(r\)\(\text{NC2}\): The proof for this case is analogous to the proof for case \(r\)\(\text{NC1}\).

Case \(r\)\(\text{NC3}\): From our rule premises we know
\[
\langle \Gamma, \Delta, [e]_r^b \varphi \rangle \xrightarrow{\alpha} \langle \Gamma, \Delta, \text{rel}(r) \text{tt} \rangle \quad (C.482)
\]
because
\[ \text{mtch}(e, a) = \bot \] (C.483)
and
\[ \Sigma; \Gamma_1 \vdash [e]_\rho \varphi \] (C.484)
\[ \Gamma = \Gamma_1 + \Gamma_2 \] (C.485)

Since \( \rho \in \{b, a\} \) we must consider two subcases:

**Subcase** \( \rho = b \): By (C.484) and tNCB we know
\[ \Sigma; \Gamma_1 \vdash \text{rel}(r) \text{tt} \] (C.486)

By (C.484) and (C.486) we know that reduction (C.482) can still be performed regardless of the contents of \( \Gamma \), hence by (C.485) we can deduce
\[ \langle \Gamma_1, \Delta, [e]_\rho \varphi \rangle \xrightarrow{\alpha} \langle \Gamma_1, \Delta, \text{rel}(r) \text{tt} \rangle \] (C.487)

\( \therefore \) Subcase holds by (C.485) and (C.487).

**Subcase** \( \rho = a \): The proof for this subcase is analogous to that of the previous subcase (i.e., \( \rho = b \)).

**Case rCN1**: From our rule premises we know
\[ \langle \Gamma, \Delta, \varphi \& \psi \rangle \xrightarrow{\alpha} \langle (\Gamma', \Gamma''), \Delta' \cup \Delta'', \varphi' \& \psi' \rangle \] (C.488)

because
\[ \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle \] (C.489)
\[ \langle \Gamma, \Delta, \psi \rangle \xrightarrow{\alpha} \langle \Gamma'', \Delta'', \psi' \rangle \] (C.490)
\[ \text{dom}(\Delta) = \text{dom}(\Delta') \cap \text{dom}(\Delta'') \] (C.491)

and
\[ \Sigma; \Gamma_1 \vdash \varphi \& \psi \] (C.492)
\[ \Gamma = \Gamma_1 + \Gamma_2 \] (C.493)

We must consider two subcases:

**Subcase** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \): Since \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \), by (C.492) and rCN2 we know
\[ \Sigma; \text{eff}(\Gamma_1, r_\psi) \vdash \varphi \] (C.494)
\[ \Sigma; \text{eff}(\Gamma_1, r_\varphi) \vdash \psi \] (C.495)
C. Proofs for Auxiliary Lemmas

Since by (C.493) we know that \( \Gamma_1 \subseteq \Gamma \), then from (C.494) and (C.495) we can deduce

\[
\Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \varphi \\
\Sigma; \text{eff}(\Gamma, r_\psi) \vdash \psi
\]  
(C.496)
(C.497)

By (C.490), (C.496) and Lemma C.3.2 we know

\[ \langle \text{eff}(\Gamma, r_\varphi), \Delta, \varphi \rangle \overset{\alpha}{\rightarrow} \langle \text{eff}(\Gamma', r_\varphi), \Delta', \varphi' \rangle \]  
(C.498)

By (C.497), (C.499) and Lemma C.3.2 we know

\[ \langle \text{eff}(\Gamma, r_\psi), \Delta, \psi \rangle \overset{\alpha}{\rightarrow} \langle \text{eff}(\Gamma'', r_\psi), \Delta'', \psi' \rangle \]  
(C.499)

By (C.493), (C.494), (C.495) and defn of eff (Def. 5.1.1) we know

\[
\text{eff}(\Gamma, r_\varphi) = \text{eff}(\Gamma_1, r_\varphi) + \Gamma_2 \\
\text{eff}(\Gamma, r_\psi) = \text{eff}(\Gamma_1, r_\psi) + \Gamma_2
\]  
(C.500)
(C.501)

By (C.494), (C.498), (C.500) and IH we know

\[ \langle \text{eff}(\Gamma_1, r_\varphi), \Delta, \varphi \rangle \overset{\alpha}{\rightarrow} \langle \text{eff}(\Gamma_1', r_\varphi), \Delta', \varphi' \rangle \]  
(C.502)

\[
\text{eff}(\Gamma', r_\varphi) = \text{eff}(\Gamma_1', r_\varphi) + \Gamma_2 \quad \Rightarrow \quad \Gamma' = \Gamma_1' + \Gamma_2
\]  
(C.503)

By (C.495), (C.499), (C.501) and IH we know

\[ \langle \text{eff}(\Gamma_1, r_\psi), \Delta, \psi \rangle \overset{\alpha}{\rightarrow} \langle \text{eff}(\Gamma_1'', r_\psi), \Delta'', \psi' \rangle \]  
(C.504)

\[
\text{eff}(\Gamma'', r_\psi) = \text{eff}(\Gamma_1'', r_\psi) + \Gamma_2 \quad \Rightarrow \quad \Gamma'' = \Gamma_1'' + \Gamma_2
\]  
(C.505)

From (C.489), (C.502) and (C.503) we know that formula \( \varphi \) can reduce into \( \varphi' \) regardless of whether we apply the side-effects or not. Hence we can deduce

\[ \langle \Gamma_1, \Delta, \varphi \rangle \overset{\alpha}{\rightarrow} \langle \Gamma_1', \Delta', \varphi' \rangle \]  
(C.506)

The same argument applies by (C.490), (C.504) and (C.505), such that we know we know

\[ \langle \Gamma_1, \Delta, \psi \rangle \overset{\alpha}{\rightarrow} \langle \Gamma_1'', \Delta'', \psi' \rangle \]  
(C.507)

By (C.491), (C.506), (C.507) and rCn1 we know

\[ \langle \Gamma_1, \Delta, \varphi \& \psi \rangle \overset{\alpha}{\rightarrow} \langle (\Gamma_1', \Gamma_1''), \Delta' \cup \Delta'', \varphi' \& \psi' \rangle \]  
(C.508)

From (C.503) and (C.505) we can conclude

\[ (\Gamma', \Gamma'') = (\Gamma_1', \Gamma_1'') + \Gamma_2 \]  
(C.509)

\[ \therefore \text{Subcase holds by (C.508) and (C.509).} \]
Subcase \( \text{excl}(\varphi,\psi) = \bot \): Since \( \text{excl}(\varphi,\psi) = \bot \), by \((C.492)\) and \(rCn1\) we know

\[
\Sigma; \Gamma_3 + \varphi \\
\Sigma; \Gamma_4 + \psi \\
\Gamma_1 = \Gamma_3 + \Gamma_4
\]

(C.510) (C.511) (C.512)

Since by \((C.512)\) we know \(\Gamma_3 \subseteq \Gamma_1\) and \(\Gamma_4 \subseteq \Gamma_1\), then by \((C.510)\) and \((C.511)\) we can deduce

\[
\Sigma; \Gamma_1 + \varphi \\
\Sigma; \Gamma_1 + \psi
\]

(C.513) (C.514)

By \((C.489), (C.493), (C.513)\) and IH we know

\[
\langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma_1', \Delta', \varphi' \rangle \\
\Gamma'' = \Gamma_1' + \Gamma_2
\]

(C.515) (C.516)

By \((C.490), (C.493), (C.514)\) and IH we know

\[
\langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma_1'', \Delta'', \psi' \rangle \\
\Gamma''' = \Gamma_1'' + \Gamma_2
\]

(C.517) (C.518)

By \((C.491), (C.515), (C.517)\) and \(rCn1\) we know

\[
\langle \Gamma_1, \Delta, \varphi \& \psi \rangle \xrightarrow{\alpha} \langle \Gamma_1', \Gamma_1'', \Delta' \cup \Delta'', \varphi' \& \psi' \rangle
\]

(C.519)

From \((C.516)\) and \((C.518)\) we can deduce

\[
(\Gamma', \Gamma'') = (\Gamma_1', \Gamma_1'') + \Gamma_2
\]

(C.520)

\[\therefore\] Subcase holds by \((C.519)\) and \((C.520)\).

Case \(rCn2\): From our rule premises we know

\[
\langle \Gamma, \Delta, \varphi \& \psi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta', \varphi' \& \psi \rangle
\]

(C.521)

because

\[
\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\tau} \langle \Gamma, \Delta', \varphi' \rangle
\]

(C.522)

and

\[
\Sigma; \Gamma_1 + \varphi \& \psi \\
\Gamma = \Gamma_1 + \Gamma_2
\]

(C.523) (C.524)

We must consider two subcases:
Subcase $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$: Since $\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi)$, by (C.523) and rCN2 we know
\[
\Sigma; \text{eff}(\Gamma_1, r_\varphi) \vdash \varphi \\
\Sigma; \text{eff}(\Gamma_1, r_\psi) \vdash \psi
\] (C.525) (C.526)
Since by (C.524) we know that $\Gamma_1 \subseteq \Gamma$, then from (C.525) we can deduce
\[
\Sigma; \text{eff}(\Gamma, r_\varphi) \vdash \varphi
\] (C.527)
By (C.522), (C.527) and Lemma C.3.2 we know
\[
\langle \text{eff}(\Gamma_1, r_\psi), \Delta, \varphi \rangle \xrightarrow{\tau} \langle \text{eff}(\Gamma, r_\psi), \Delta', \varphi' \rangle
\] (C.528)
By (C.524), (C.525) and the defn of eff (Def. 5.1.1) we know
\[
\text{eff}(\Gamma, r_\psi) = \text{eff}(\Gamma_1, r_\psi) + \Gamma_2
\] (C.529)
By (C.525), (C.528), (C.529) and IH we know
\[
\langle \text{eff}(\Gamma_1, r_\psi), \Delta, \varphi \rangle \xrightarrow{\tau} \langle \text{eff}(\Gamma_1, r_\psi), \Delta', \varphi' \rangle
\] (C.530)
\[
\text{eff}(\Gamma, r_\psi) = \text{eff}(\Gamma_1, r_\psi) + \Gamma_2 \implies \Gamma = \Gamma_1 + \Gamma_2
\] (C.531)
From (C.522), (C.530) and (C.531) we know that formula $\varphi$ can reduce into $\varphi'$ using subset $\Gamma_1$, as the reduction can still be applied regardless of whether we apply the side-effects or not. Hence we can deduce
\[
\langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\tau} \langle \Gamma_1, \Delta', \varphi' \rangle
\] (C.532)
By (C.532) and rCN2 we know
\[
\langle \Gamma_1, \Delta, \varphi \& \psi \rangle \xrightarrow{\tau} \langle \Gamma_1, \Delta', \varphi' \& \psi \rangle
\] (C.533)
\therefore: Subcase holds by (C.531) and (C.533).

Subcase $\text{excl}(\varphi, \psi) = \bot$: Since $\text{excl}(\varphi, \psi) = \bot$, by (C.523) and rCN1 we know
\[
\Sigma; \Gamma_3 \vdash \varphi \\
\Sigma; \Gamma_4 \vdash \psi
\] (C.534) (C.535)
\[
\Gamma_1 = \Gamma_3 + \Gamma_4
\] (C.536)
Since by (C.536) we know $\Gamma_3 \subseteq \Gamma_1$, then from (C.534) we can deduce
\[
\Sigma; \Gamma_1 \vdash \varphi
\] (C.537)
By (C.522), (C.524), (C.537) and IH we know
\[
\langle \Gamma_1, \Delta, \varphi \rangle \xrightarrow{\tau} \langle \Gamma_1, \Delta', \varphi' \rangle
\] (C.538)
\[
\Gamma = \Gamma_1 + \Gamma_2
\] (C.539)
By (C.538) and rCn2 we know
\[ \langle \Gamma_1, \Delta, \varphi \& \psi \rangle \xrightarrow{r} \langle \Gamma_1, \Delta', \varphi' \& \psi \rangle \] (C.540)
\[ \therefore \text{Subcase holds by (C.539) and (C.540).} \]

**Case rCn3:** The proof for this case is analogous to the proof of case rCn2. \[ \square \]

**To Prove Lemma C.3.4.**

\[ \text{excl}(\varphi,\psi) = (r_\varphi, r_\psi) \text{ and } \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{r} \langle \Gamma, \Delta', \varphi' \rangle \implies \text{excl}(\varphi',\psi) = (r_\varphi, r_\psi). \]

**Proof.** By rule induction on \( \langle \Gamma, \Delta, \varphi \rangle \xrightarrow{r} \langle \Gamma, \Delta', \varphi' \rangle \).

**Case rTru:** From our rule premises we know
\[ \langle \Gamma, \Delta, \text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2 \rangle \xrightarrow{r} \langle \Gamma, \Delta, \varphi_1 \rangle \] (C.541)
\[ \text{excl}(\text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2, \psi) = (r_\varphi, r_\psi) \] (C.542)

By (C.542) and defn of excl (Def. 5.1.2) we know
\[ \text{excl}(\varphi, \psi) = \text{excl}(\varphi_2, \psi) = (r_\varphi, r_\psi) \] (C.543)
\[ \therefore \text{Case holds by (C.543).} \]

**Case rFls:** The proof for this case is analogous to the proof for rTru.

**Case rMax:** From our rule premises we know
\[ \langle \Gamma, \Delta, \text{max } X. \varphi \rangle \xrightarrow{r} \langle \Gamma, \Delta, \varphi[\text{clr}(X)\text{max } X. \varphi]/X \rangle \] (C.544)
\[ \text{excl} (\text{max } X. \varphi, \psi) = (r_\varphi, r_\psi) \] (C.545)

By (C.545) and defn of excl (Def. 5.1.2) we know
\[ \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \] (C.546)

By [(clr(X)max X.\varphi)/X], (C.546) and Lemma C.3.8 we know
\[ \text{excl}(\varphi[\text{clr}(X)\text{max } X. \varphi]/X, \psi) = (r_\varphi, r_\psi) \] (C.547)
\[ \therefore \text{Case holds by (C.547).} \]

**Case rClr:** The proof for this case is analogous to the proof for rMax.

208
C. Proofs for Auxiliary Lemmas

Case rCn2: From our rule premises we know

$$
\langle \Gamma, \Delta, \varphi_1 & \varphi_2 \rangle \xrightarrow{\tau} \langle \Gamma', \Delta', \varphi'_1 & \varphi'_2 \rangle
$$

(C.548)

because

$$
\langle \Gamma, \Delta, \varphi_1 \rangle \xrightarrow{\tau} \langle \Gamma, \Delta', \varphi'_1 \rangle
$$

(C.549)

and

$$
excl((\varphi_1 & \varphi_2), \psi) = (r_{(\varphi_1 & \varphi_2)}, r_{\psi})
$$

(C.550)

By (C.550) and defn of excl (Def. 5.1.2) we know

$$
excl(\varphi_1, \psi) = (r_{\varphi_1}, r_{\psi})
$$

(C.551)

$$
excl(\varphi_2, \psi) = (r_{\varphi_2}, r_{\psi})
$$

(C.552)

$$
r_{(\varphi_1 & \varphi_2)} = r_{\varphi_1} \cup r_{\varphi_2}
$$

(C.553)

By (C.549), (C.551) and IH we know

$$
excl(\varphi'_1, \psi) = (r_{\varphi'_1}, r_{\psi})
$$

(C.554)

By (C.552), (C.553), (C.554) and defn of excl (Def. 5.1.2) we know

$$
excl((\varphi'_1 & \varphi_2), \psi) = (r_{(\varphi'_1 & \varphi_2)}, r_{\psi})
$$

(C.555)

∴ Case holds by (C.555).

Note: The remaining cases do not apply as we only consider \(\tau\) actions.

To Prove Lemma C.3.5. excl(\(\varphi, \psi\)) = (r_{\varphi}, r_{\psi}) and \(\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle\) and

$$
\Sigma; \text{eff}((\Gamma', \Gamma''), r_{\psi}) + \varphi' \quad \text{and} \quad \Sigma; \text{eff}((\Gamma', \Gamma''), r_{\varphi}) + \psi' \quad \text{implies} \quad \Sigma; (\Gamma', \Gamma'') + \varphi' & \psi'.
$$

Proof. By rule induction on \(\langle \Gamma, \Delta, \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \varphi' \rangle\).

Case rNc1: From our rule premises we know

$$
\langle \Gamma, \Delta, [e_{\alpha}]^b \varphi \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta', \text{blk}(i) \varphi \sigma \rangle
$$

(C.556)

because

$$
\text{mtch}(e, \alpha) = \sigma
$$

(C.557)

and others . . .
and

\[
\text{excl}(\lbrack e \rbrack b \varphi, \psi) = (r, r_\varphi) \tag{C.558}
\]

\[
\Sigma; \text{eff}((\Gamma', \Gamma''), r_\varphi) \vdash \text{blk}(i) \varphi \sigma 
\]

\[
\Sigma; \text{eff}((\Gamma', \Gamma''), r) \vdash \psi' \tag{C.560}
\]

By (C.558) and defn of excl (Def. 5.1.2) we know

\[
\forall e_1 \in \text{fps}(\psi) \cdot \text{mtch}(e, e_1) = \bot \tag{C.561}
\]

By (C.557) we know that system event \( \alpha \) matches pattern \( e \), but by (C.561) we also know that \( \alpha \) does not match any of the concurrent necessities defined in \( \psi \). This means that all the other necessities defined in \( \psi \) are trivially satisfied. Hence we can deduce

\[
\psi' = \&_{n=1} (\text{rel}(r) n)_n \tag{C.562}
\]

Since \( \lbrack e \rbrack b \varphi \) reduces into \( \text{blk}(i) \varphi \sigma \), by (C.562) and defn of excl (Def. 5.1.2) we know

\[
\text{excl}(\text{blk}(i) \varphi \sigma, \psi') = (\varepsilon, r_\psi) \tag{C.563}
\]

Since \( \varepsilon \subseteq r \), and by defn of eff (Def. 5.1.1), we know that (C.560) still typechecks using \( \varepsilon \) instead of \( r \) given that we know (C.562). (Remember that the eff function “releases” a number of blocked ids from a type environment \( \Gamma \), so if we don’t release them \( \psi' = \&_{n=1} (\text{rel}(r) n)_n \) still typechecks). Hence we can deduce

\[
\Sigma; \text{eff}((\Gamma', \Gamma''), \varepsilon) \vdash \psi' \tag{C.564}
\]

By (C.559), (C.563), (C.564) and \( \tau \text{Cn2} \) we know

\[
\Sigma; (\Gamma', \Gamma'') \vdash \text{blk}(i) \varphi \sigma \& \psi' \tag{C.565}
\]

\[\therefore \text{Case holds by (C.565).}\]

**Case rNc2:** The proof for this case is analogous to that of case rNc1.

**Case rNc3:** From our rule premises we know

\[
\langle \Gamma, \Delta, \lbrack e_2 \rbrack b \varphi \rangle \xrightarrow{\alpha} \langle \Gamma, \Delta, \text{rel}(r) t t \rangle \tag{C.566}
\]

because

\[
\text{mtch}(e, \alpha) = \bot \tag{C.567}
\]
C. Proofs for Auxiliary Lemmas

and

\[ \text{excl}([e]^p_\varphi, \psi) = (r, r_{\varphi}) \quad \text{(C.568)} \]
\[ \Sigma; \text{eff}((\Gamma', \Gamma''), r_{\varphi}) \vdash \text{rel}(r) \text{ tt} \quad \text{(C.569)} \]
\[ \Sigma; \text{eff}((\Gamma', \Gamma''), r) \vdash \psi' \quad \text{(C.570)} \]

By (C.568) and defn of \text{excl} (Def. 5.1.2) we know

\[ \forall e_1 \in \text{fps}(\psi) \cdot \text{mtch}(e, e_1) = \bot \quad \text{(C.571)} \]

By (C.567) we know that system event \( \alpha \) does not match pattern \( e \), and by (C.571) we know that the opposing branch \( \psi \) may have lived on. We must therefore consider these two cases:

**Subcase** \( \psi \) also died \(-\psi = \&_{\text{rel}((r)n)}^k \): Since \([e_1]^p_\varphi \) reduces to \( \text{rel}(r) \text{ tt} \), \( \psi = \&_{\text{rel}((r)n)}^k \), and by defn of \text{excl} (Def. 5.1.2) we know

\[ \text{excl}(\text{rel}(r) \text{ tt}, \psi') = (r, r_{\varphi}) \quad \text{(C.572)} \]

By (C.569), (C.570), (C.572) and \( \tau\text{C}n2 \) we know

\[ \Sigma; (\Gamma', \Gamma'') \vdash \text{rel}(r) \text{ tt} \land \psi' \quad \text{(C.573)} \]

\( \therefore \) Subcase holds by (C.573).

**Subcase** \( \psi \) lived on \(-\psi \neq \&_{\text{rel}((r)n)}^k \): Since \([e_1]^p_\varphi \) reduces to \( \text{rel}(r) \text{ tt} \neq \&_{\text{rel}((r)n)}^k \), and by defn of \text{excl} (Def. 5.1.2) we know

\[ \text{excl}(\text{rel}(r) \text{ tt}, \psi') = (r, \varepsilon) \quad \text{(C.574)} \]

Since \( \varepsilon \subseteq r_{\varphi} \), and by defn of \text{eff} (Def. 5.1.1), we know that (C.569) still typechecks using \( \varepsilon \) instead of \( r_{\varphi} \) given that we know (C.574). Hence we can deduce

\[ \Sigma; \text{eff}((\Gamma', \Gamma''), \varepsilon) \vdash \text{rel}(r) \text{ tt} \quad \text{(C.575)} \]

By (C.570), (C.574), (C.575) and \( \tau\text{C}n2 \) we know

\[ \Sigma; (\Gamma', \Gamma'') \vdash \text{rel}(r) \text{ tt} \land \psi' \quad \text{(C.576)} \]

\( \therefore \) Subcase holds by (C.576).

**Case** \( \text{rC}n1 \): From our rule premises we know

\[ \langle \Gamma, \Delta, \varphi_1 \land \varphi_2 \rangle \xrightarrow{\alpha} \langle \Gamma', \Delta' \cup \Delta'', \varphi'_1 \land \varphi'_2 \rangle \quad \text{(C.577)} \]
\[ \text{excl}(\varphi_1 \land \varphi_2, \psi) = (r_{\varphi}, r_{\varphi}) \quad \text{(C.578)} \]
\[ \Sigma; \text{eff}((\Gamma', \Gamma''), r_{\varphi}) \vdash \varphi'_1 \land \varphi'_2 \quad \text{(C.579)} \]
\[ \Sigma; \text{eff}((\Gamma', \Gamma''), r_{\varphi}) \vdash \psi' \quad \text{(C.580)} \]
As by (C.578) we know that branch \( \varphi_1 \& \varphi_2 \) is mutually exclusive to \( \psi \), then we have to consider the following three subcases:

**Subcase** \( \varphi_1' \& \varphi_2' = \&^{k}_{n=1} (r_\varphi)_n \) and \( \psi' \neq \&^{k}_{n=1} (r_\varphi)_n \): As we assume that \( \varphi_1 \& \varphi_2 \) reduces into \( \&^{k}_{n=1} (r_\varphi)_n \), by (C.578) and defn of excl (Def. 5.1.2) we know

\[
\text{excl}(\&^{k}_{n=1} (r_\varphi)_n, \psi') = (r_\varphi, r_\psi)
\]

(C.581)

Since \( \varphi_1' \& \varphi_2' = \&^{k}_{n=1} (r_\varphi)_n \), and defn of eff (Def. 5.1.1) we know that (C.579) still holds if \( r_\varphi = \varepsilon \). (Remember that the eff function “releases” a number of blocked ids from a type environment \( \Gamma \), so if we don’t release them \( \psi' = \&^{k}_{n=1} (r_\varphi)_n \) still typechecks). Hence we can deduce

\[
\Sigma; \text{eff}((\Gamma', \Gamma''), \varepsilon) \vdash \varphi_1' \& \varphi_2'
\]

(C.582)

By (C.580), (C.581), (C.582) and \( \tau \text{Cn}2 \) we know

\[
\Sigma; (\Gamma', \Gamma'') \vdash (\varphi_1' \& \varphi_2') \& \psi'
\]

(C.583)

\( \therefore \) Subcase holds by (C.583).

**Subcase** \( \psi' = \&^{k}_{n=1} (r_\varphi)_n \) and \( \varphi_1' \& \varphi_2' = \&^{k}_{n=1} (r_\varphi)_n \): The proof for this subcase is analogous to that of the previous subcase.

**Subcase** \( \varphi_1' \& \varphi_2' = \&^{k}_{n=1} (r_\varphi)_n \) and \( \psi' = \&^{k}_{n=1} (r_\varphi)_n \): As we assume that both branches \( \varphi_1 \& \varphi_2 \) and \( \psi \) are trivially satisfied, i.e., reduce into the form \( \&^{k}_{n=1} (r_\varphi)_n \), by (C.578) and defn of excl (Def. 5.1.2) we know

\[
\text{excl}(\varphi_1' \& \varphi_2', \psi') = (r_\varphi, r_\psi)
\]

(C.584)

By (C.579), (C.580), (C.584) and \( \tau \text{Cn}2 \) we know

\[
\Sigma; (\Gamma', \Gamma'') \vdash (\varphi_1' \& \varphi_2') \& \psi'
\]

(C.585)

\( \therefore \) Subcase holds by (C.585).

**Note:** The remaining cases do not apply as we only consider \( \alpha \) actions. \( \square \)

**To Prove Lemma C.3.6.** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \( (\Gamma, \Delta, \varphi) \xrightarrow{\mu} (\Gamma', \Delta, \varphi') \) and

\[
\Sigma; \text{eff}(\Gamma', r_\varphi) \vdash \psi \quad \text{and} \quad \Sigma; \text{eff}(\text{after}(\Gamma, \mu), r_\psi) \vdash \varphi' \quad \text{implies} \quad \Sigma; \text{after}(\Gamma, \mu) \vdash \varphi' \& \psi
\]

**Proof.** By rule induction on \( (\Gamma, \Delta, \varphi) \xrightarrow{\mu} (\Gamma', \Delta, \varphi') \).
C. Proofs for Auxiliary Lemmas

**Case rBlk:** From our rule premises we know

\[
\langle (\Gamma, w: \text{lid}), \Delta, \text{blk}(w) \varphi \rangle \xrightarrow{b(w)} \langle (\Gamma, w: \text{lbid}), \Delta, \varphi \rangle \quad (C.586)
\]

\[
\text{excl}(\text{blk}(w) \varphi, \psi) = (\varepsilon, r_{\varphi}) \quad (C.587)
\]

\[
\Sigma; \text{eff}(\text{after}((\Gamma, w: \text{lid}), b(w)), r_{\varphi}) \vdash \varphi \quad (C.588)
\]

\[
\Sigma; \text{eff}(\langle (\Gamma, w: \text{lid}), \varepsilon \rangle, r_{\psi}) \vdash \psi \quad (C.589)
\]

By (C.588) and the defn of after (Def. 5.3.2) we know

\[
\Sigma; \text{eff}(\langle (\Gamma, w: \text{lbid}), r_{\varphi} \rangle) \vdash \varphi \quad (C.590)
\]

We know that by the defn of excl (Def. 5.1.2), (C.587) can only hold if \(\psi\) represents a collection of trivially satisfied branches, hence we can deduce

\[
\text{excl}(\varphi, \psi) = (\varepsilon, r_{\varphi}) \quad (C.591)
\]

\[
\psi = \delta_{\text{mb}}^{k}(\text{rel}(r)n)_{n} \quad (C.592)
\]

By (C.592) we know that for (C.589) to hold, then every identifier in the release set \((r_{\varphi})\) of \(\psi\) must be of the type \(\text{lbid}\). Since we know that \(w: \text{lid}\), then we know for sure that release-set \(r_{\varphi}\) does not intersect with blocking set \(w\) (otherwise \(C.589\) would not hold). Hence we know that \(\psi\) can typecheck regardless of the type of \(w\) (i.e., it typechecks both if we have \(w: \text{lid}\) or \(w: \text{lbid}\)), such that we can conclude

\[
\Sigma; \text{eff}(\langle (\Gamma, w: \text{lbid}), \varepsilon \rangle) \vdash \psi \quad (C.593)
\]

By (C.590), (C.591), (C.593), \(\tau\text{Cn}2\) and the defn of after (Def. 5.3.2) we know

\[
\Sigma; \text{after}((\Gamma, w: \text{lid}), b(w)) \vdash \varphi \& \psi \quad (C.594)
\]

\(\therefore\) Case holds by (C.594).

**Case rRel:** From our rule premises we know

\[
\langle (\Gamma, r: \text{lbid}), \Delta, \text{rel}(r) \varphi \rangle \xrightarrow{r(\cdot)} \langle (\Gamma, r: \text{lid}), \Delta, \varphi \rangle \quad (C.595)
\]

\[
\text{excl}(\text{rel}(r) \varphi, \psi) = (r_{\varphi}, r_{\psi}) \quad (C.596)
\]

\[
\Sigma; \text{eff}(\text{after}((\Gamma, r: \text{lbid}), \text{r}(r)), r_{\varphi}) \vdash \varphi \quad (C.597)
\]

\[
\Sigma; \text{eff}(\langle (\Gamma, r: \text{lbid}), r \rangle) \vdash \psi \quad (C.598)
\]

By (C.596) and the defn of excl (Def. 5.1.2) we must consider 2 subcases:

**Subcase** \(\varphi \neq \text{tt}\) and \(\psi = \delta_{\text{mb}}^{k}(\text{rel}(r)n)_{n}\): The proof for this subcase is analogous to the proof of case rBlk.
C. Proofs for Auxiliary Lemmas

Subcase $\varphi = \tt$: Since $\varphi = \tt$, by (C.596), (C.597) and (C.598) we know

\[ \text{excl}(\text{rel}(r) \tt, \psi) = (r, r_\varphi) \]  
(C.599)

\[ \Sigma; \text{eff}(\text{after}((\Gamma, r : \text{lid}), \text{r}(r)), r_\varphi) \vdash \tt \]
(C.600)

\[ \Sigma; \text{eff}((\Gamma, r : \text{lid}), r) \vdash \psi \]  
(C.601)

By (C.599) and the defn of excl (Def. 5.1.2) we know

\[ \text{excl}(\tt, \psi) = (\varepsilon, r_\psi) \]  
(C.602)

By (C.600) and the defn of after (Def. 5.3.2) we know

\[ \Sigma; \text{eff}(((\Gamma, r : \text{lid}), r_\varphi) \vdash \tt \]  
(C.603)

By (C.601) and the defn of eff (Def. 5.1.1) we know

\[ \Sigma; \text{eff}((\Gamma, r : \text{lid}), \varepsilon) \vdash \psi \]  
(C.604)

By (C.602), (C.603), (C.604), $\tau Cn2$ and the defn of after (Def. 5.3.2) we know

\[ \Sigma; \text{after}((\Gamma, r : \text{lid}), \text{r}(r)) \vdash \tt \& \psi \]  
(C.605)

∴ Case holds by (C.605).

Case $rA\text{dS}$: From our rule premises we know

\[ (((\Gamma, w : \text{lid}), \Delta, \text{sA}(w), \varphi \Downarrow (w)) \xrightarrow{\text{s}(w)} ((\Gamma, w : \text{lid}), \Delta, \text{rel}(r) \varphi) \]  
(C.606)

\[ \text{excl}(\text{rel}(r) \varphi, \psi) = (\varepsilon, r_\varphi) \]  
(C.607)

\[ \Sigma; \text{eff}(\text{after}((\Gamma, w : \text{lid}), \text{s}(w)), r_\varphi) \vdash \text{rel}(r) \varphi \]
(C.608)

\[ \Sigma; \text{eff}((\Gamma, w : \text{lid}), \varepsilon) \vdash \psi \]  
(C.609)

By (C.607) and the defn of excl (Def. 5.1.2) we know

\[ \psi = \&^{\text{rel}(r)n}_n \]  
(C.610)

By (C.608) and the defn of after (Def. 5.3.2) we know

\[ \Sigma; \text{eff}((\Gamma, w : \text{lid}), r_\varphi) \vdash \text{rel}(r) \varphi \]  
(C.612)

By (C.610), (C.609), (C.612), $\tau Cn2$ and the defn of after (Def. 5.3.2) we know

\[ \Sigma; \text{after}((\Gamma, w : \text{lid}), \text{s}(w)) \vdash \text{rel}(r) \varphi \& \psi \]  
(C.613)

∴ Case holds by (C.613).
C. Proofs for Auxiliary Lemmas

**Case rAdA:** The proof for this case is analogous to the proof for case rAdS.

**Case rCn3:** From our rule premises we know

\[
(\Gamma, \Delta, \varphi_1 \& \varphi_2) \xrightarrow{\mu} (\Gamma', \Delta, \varphi_1' \& \varphi_2) \tag{C.614}
\]

\[\text{excl}(\varphi_1 \& \varphi_2, \psi) = (r_\varphi, r_\psi) \tag{C.615}\]

\[\Sigma; \text{eff}(\text{after}(\Gamma, \mu), r_\psi) \vdash \varphi_1' \& \varphi_2 \tag{C.616}\]

\[\Sigma; \text{eff}(\Gamma, r_\psi) \vdash \psi \tag{C.617}\]

As by \text{(C.615)} we know that branch \(\varphi_1 \& \varphi_2\) is mutually exclusive to \(\psi\), based on the defn of excl (Def. 5.1.2) we have to consider the following three subcases:

**Subcase** \(\varphi_1 \& \varphi_2 = \&^{k}_{\text{rel}}(\text{rel}(r)n)_n\) and \(\psi \neq \&^{k}_{\text{rel}}(\text{rel}(r)n)_n\): By our subcase assumptions and \text{(C.614)} we can deduce that the reduction can only be made to issue a release of one of the releases sets declared in the collection of trivially satisfied branches \(\varphi_1 \& \varphi_2\). Hence we can conclude

\[
\Gamma = \Gamma'', r_\varphi : \text{lid} \tag{C.618}\]

\[\mu = r_\varphi \tag{C.619}\]

\[r_\varphi = r_{\varphi_1}, r_{\varphi_{n-1}} \tag{C.620}\]

\[\varphi_1 \& \varphi_2 = (\text{rel}(r_\varphi) \& \&^{k}_{\text{rel}}(\text{rel}(r)n)_n) \tag{C.621}\]

\[\varphi_1' \& \varphi_2 = \&^{k}_{\text{rel}}(\text{rel}(r)n)_n \tag{C.622}\]

where \(r_{\varphi_1}\) is the release set of an arbitrary trivially satisfied branch \(\text{rel}(r_{\varphi_1})\).

By \text{(C.615)}, \text{(C.620)}, \text{(C.621)}, \text{(C.622)} and the defn of excl (Def. 5.1.2) we know

\[\text{excl}(\varphi_1' \& \varphi_2, \psi) \equiv \text{excl}(\&^{k}_{\text{rel}}(\text{rel}(r)n)_n, \psi) = (r_{\varphi_{n-1}}, \varepsilon) \tag{C.623}\]

Since \(\psi \neq \&^{k}_{\text{rel}}(\text{rel}(r)n)_n\), by \text{(C.616)}, \text{(C.617)}, \text{(C.618)} and \text{(C.619)} we can deduce

\[\Sigma; \text{eff}(\text{after}(\Gamma'', r_\varphi : \text{lid}, (r(r_\varphi))), \varepsilon) \vdash \varphi_1' \& \varphi_2 \tag{C.624}\]

\[\Sigma; \text{eff}(\Gamma'', r_\varphi : \text{lid}, r_\psi) \vdash \psi \tag{C.625}\]

By \text{(C.624)}, \text{(C.620)} and the defn of after (Def. 5.3.2) we know

\[\Sigma; \text{eff}(((\Gamma''', r_{\varphi_{n-1}} : \text{lid}, r_{\varphi_1} : \text{lid}), \varepsilon) \vdash \varphi_1' \& \varphi_2 \tag{C.626}\]

By \text{(C.625)}, \text{(C.620)} and the defn of eff (Def. 5.1.1) we know

\[\Sigma; \text{eff}((\Gamma''', r_{\varphi_{n-1}} : \text{lid}, r_{\varphi_1} : \text{lid}, r_{\varphi_{n-1}}) \vdash \psi \tag{C.627}\]

By \text{(C.623)}, \text{(C.626)}, \text{(C.627)}, \tau\text{Cn2} and the defn of after (Def. 5.3.2) we know

\[\Sigma; (\Gamma'', r_{\varphi_{n-1}} : \text{lid}, r_{\varphi_1} : \text{lid}) \vdash (\varphi_1' \& \varphi_2) \& \psi \equiv \Sigma; (\text{after}((\Gamma''', r_{\varphi_{n-1}} : \text{lid}, r(r_{\varphi_1}))) \vdash (\varphi_1' \& \varphi_2) \& \psi \tag{C.628}\]
C. Proofs for Auxiliary Lemmas

.: Subcase holds by (C.629).

**Subcase** \( \varphi_1 \& \varphi_2 = \&_{n=1}^{k} (\text{re}(r))_n \) and \( \psi = \&_{n=1}^{k} (\text{re}(r))_n \): The proof for this subcase follows a similar argument as per the previous subcase.

**Subcase** \( \varphi_1 \& \varphi_2 \neq \&_{n=1}^{k} (\text{re}(r))_n \) and \( \psi = \&_{n=1}^{k} (\text{re}(r))_n \): As we assume that \( \psi = \&_{n=1}^{k} (\text{re}(r))_n \), if \( \varphi_1 \) commits an action \( \mu \in \{b(r_1), r(r_1), \ldots\} \) and becomes \( \varphi'_1 \), then we are certain that \( r \) does not conflict with any release set in pertaining to \( \psi \), as otherwise \( \text{(C.617)} \) would not hold. Therefore we can deduce that after committing an action \( \mu \), \( \psi = \&_{n=1}^{k} (\text{re}(r))_n \) would not be effected, such that we can conclude

\[
\begin{align*}
\Sigma; \text{eff}(after(\Gamma, \mu), e) &\vdash \psi \\
\\text{excl}(\varphi'_1 \& \varphi_2, \psi) &= (e, r_\psi)
\end{align*}
\]

By (C.616), (C.629), (C.630) and \( \tau \text{Cn}2 \) we conclude

\[
\Sigma; (after(\Gamma, \mu)) + (\varphi'_1 \& \varphi_2) \& \psi
\]

.: Subcase holds by (C.631).

**Note:** The remaining cases do not apply as we only consider \( \mu \) actions. \( \Box \)

**To Prove Lemma C.3.7.** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \([\text{clr}(X) \text{max } X. \varphi')/X\] implies \( \text{excl}(\varphi[\text{clr}(X) \text{max } X. \varphi')/X], \psi[\text{clr}(X) \text{max } X. \varphi')/X]) = (r_\varphi, r_\psi) \)

**Proof.** From our premises we know

\[
\begin{align*}
\\text{excl}(\varphi, \psi) &= (r_\varphi, r_\psi) \\
[\text{clr}(X) \text{max } X. \varphi')/X] &= \text{(C.632)} \\
[\text{clr}(X) \text{max } X. \varphi')/X] &= \text{(C.633)}
\end{align*}
\]

By (C.632), (C.633) and Lemma [C.3.8] we know

\[
\text{excl}(\varphi[\text{clr}(X) \text{max } X. \varphi')/X], \psi) = (r_\varphi, r_\psi)
\]

(C.634)

By (C.634) and symmetry we know

\[
\text{excl}(\psi, \varphi[\text{clr}(X) \text{max } X. \varphi')/X]) = (r_\psi, r_\varphi)
\]

(C.635)

By (C.633), (C.635) and Lemma [C.3.8] we know

\[
\text{excl}(\psi[\text{clr}(X) \text{max } X. \varphi')/X], \varphi[\text{clr}(X) \text{max } X. \varphi')/X]) = (r_\psi, r_\varphi)
\]

(C.636)

By (C.636) and symmetry we know

\[
\text{excl}(\varphi[\text{clr}(X) \text{max } X. \varphi')/X], \psi[\text{clr}(X) \text{max } X. \varphi')/X]) = (r_\varphi, r_\psi)
\]

(C.637)

.: Lemma holds by (C.637).

**To Prove Lemma C.3.8.** \( \text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \) and \([\text{clr}(X) \text{max } X. \varphi')/X\] implies \( \text{excl}(\varphi[\text{clr}(X) \text{max } X. \varphi')/X], \psi) = (r_\varphi, r_\psi) \).
C. Proofs for Auxiliary Lemmas

Proof. By structural induction on \( \varphi \).

Case \( \varphi = \max Y. \varphi \): From our rule premises we know

\[
\text{excl}(\max Y. \varphi, \psi) = (r_\varphi, r_\psi) \quad \text{ (C.638)}
\]
\[
[(\text{clr}(X)\max X. \varphi')/X] \quad \text{ (C.639)}
\]

We must consider two subcases.

Subcase \( \psi \neq \&_{\text{max}}^k (\text{rol}(r)n)_n \): By (C.638) and defn of excl (Def. 5.1.2) we know

\[
\text{excl}(\varphi, \psi) = (r_\varphi, r_\psi) \quad \text{ (C.640)}
\]

By (C.639), (C.640) and IH we know

\[
\text{excl}([(\text{clr}(X)\max X. \varphi')/X], \psi) = (r_\varphi, r_\psi) \quad \text{ (C.641)}
\]

By (C.641) and defn of excl (Def. 5.1.2) we can deduce

\[
\text{excl}((\max Y. \varphi)[(\text{clr}(X)\max X. \varphi')/X]), \psi) = (r_\varphi, r_\psi) \quad \text{ (C.642)}
\]

By (C.642) and substitution we know

\[
\text{excl}((\max Y. \varphi)[(\text{clr}(X)\max X. \varphi')/X], \psi) = (r_\varphi, r_\psi) \quad \text{ (C.643)}
\]

\( \therefore \) Subcase holds by (C.643).

Subcase \( \psi = \&_{\text{max}}^k (\text{rol}(r)n)_n \): By defn of excl (Def. 5.1.2) and since \( \psi = \&_{\text{max}}^k (\text{rol}(r)n)_n \) we know that no matter what \( \varphi \) is, the excl function always returns \( \varepsilon \) as the release set of \( \varphi \). Hence by (C.638) and (C.639) we know

\[
\text{excl}((\max Y. \varphi), \&_{\text{max}}^k (\text{rol}(r)n)_n) = \text{excl}((\max Y. \varphi)[(\text{clr}(X)\max X. \varphi')/X], \&_{\text{max}}^k (\text{rol}(r)n)_n) = (\varepsilon, r_\psi) \quad \text{ (C.644)}
\]

\( \therefore \) Subcase holds by (C.644).

Case \( \varphi = \text{clr}(Y)\varphi \): The proof for this case is analogous to that of case \( \varphi = \max Y. \varphi \).

Case \( \varphi = \varphi_1 \& \varphi_2 \): From our rule premises we know

\[
\text{excl}((\varphi_1 \& \varphi_2), \psi) = (r_\varphi, r_\psi) \quad \text{ (C.645)}
\]
\[
[(\text{clr}(X)\max X. \varphi')/X] \quad \text{ (C.646)}
\]
C. Proofs for Auxiliary Lemmas

By (C.645) and defn of excl (Def. 5.1.2) we know
\[
excl(\varphi_1, \psi) = (r_{\varphi_1}, r_\psi) \quad (C.647)
\]
\[
excl(\varphi_2, \psi) = (r_{\varphi_2}, r_\psi) \quad (C.648)
\]
\[
r_\varphi = (r_{\varphi_1} \cup r_{\varphi_2}) \quad (C.649)
\]
By (C.646), (C.647) and IH we know
\[
excl(\varphi_1[\{(\text{clr}(X)\text{max }X. \varphi')/X\}, \psi) = (r_{\varphi_1}, r_\psi) \quad (C.650)
\]
By (C.646), (C.648) and IH we know
\[
excl(\varphi_2[\{(\text{clr}(X)\text{max }X. \varphi')/X\}, \psi) = (r_{\varphi_2}, r_\psi) \quad (C.651)
\]
By (C.649), (C.650), (C.651) and defn of excl (Def. 5.1.2) we know
\[
excl(\varphi_1[\{(\text{clr}(X)\text{max }X. \varphi')/X\}, \psi) = (r_{\varphi_1}, r_\psi) \quad (C.652)
\]
By (C.652) and substitution we know
\[
excl(\varphi_1 \& \varphi_2[\{(\text{clr}(X)\text{max }X. \varphi')/X\}, \psi) = (r_{\varphi_1}, r_\psi) \quad (C.653)
\]
\[\therefore \text{ Case holds by (C.653).}\]

Case \(\varphi = [e]^\varphi\): From our rule premises we know
\[
excl([e]^\varphi, \psi) = (r_{\varphi}, r_\psi) \quad (C.654)
\]
\[
[\{(\text{clr}(X)\text{max }X. \varphi')/X\}] \quad (C.655)
\]
We must consider two subcases.

Subcase \(\psi \neq \&_{\text{rel}(r)\text{n}}\): By (C.654) and defn of excl (Def. 5.1.2) we know
\[
excl(\text{rel}(r) \text{tt}, \psi) = (r, r_\psi) \quad (C.656)
\]
\[
\forall e_1 \in \text{fps}(\psi) \cdot \text{mtch}(e, e_1) = \bot \quad (C.657)
\]
Since \(\text{rel}(r) \text{ tt} \equiv (\text{rel}(r) \text{ tt})[\{(\text{clr}(X)\text{max }X. \varphi')/X\}]\), by (C.656) we know
\[
excl((\text{rel}(r) \text{ tt})[\{(\text{clr}(X)\text{max }X. \varphi')/X\}], \psi) = (r, r_\psi) \quad (C.658)
\]
By (C.657), (C.658) and defn of excl (Def. 5.1.2) we can deduce
\[
excl([e]^\varphi)[\{(\text{clr}(X)\text{max }X. \varphi')/X\}], \psi) = (r, r_\psi) \quad (C.659)
\]
\[\therefore \text{ Subcase holds by (C.659).}\]

Subcase \(\psi = \&_{\text{rel}(r)\text{n}}\): By defn of excl (Def. 5.1.2) and since \(\psi = \&_{\text{rel}(r)\text{n}}\) we know that no matter what \(\varphi\) is, the excl function always returns \(\varepsilon\) as the release set of \(\varphi\).
C. Proofs for Auxiliary Lemmas

Hence by (C.654) and (C.655) we know

\[
\text{excl}(\varepsilon, \rho, r, \phi, k_n = 1) = (\text{rel}(r))_{\text{tt}}^{n} (X, \phi')/X, \text{excl}(\varepsilon, r, \phi) \tag{C.660}
\]

\[
\therefore \text{Subcase holds by (C.660).}
\]

**Case** $\phi = sA(r)w\phi$: From our rule premises we know

\[
\text{excl}(sA(r)w, \phi, \psi) = (r, r, r, \psi) \tag{C.661}
\]

\[
\text{excl}(\varepsilon, r, \psi) \tag{C.662}
\]

By defn of excl (Def. 5.1.2) we know that (C.654) can only be true if $\psi = \text{excl}(r)_{\text{tt}}^{n}$ (because otherwise, excl would have returned $\bot$). Hence we know

\[
\psi = \text{excl}(r)_{\text{tt}}^{n} \tag{C.663}
\]

Since we know (C.663), then we also know that no matter what $\phi$ is, the excl function always returns $\varepsilon$ as the release set of $\phi$. Hence by (C.661), (C.662) and defn of excl (Def. 5.1.2) we can deduce

\[
\text{excl}(sA(r)w, \phi, \psi) = (r, r, r, \psi) \tag{C.664}
\]

\[
\therefore \text{Case holds by (C.664).}
\]

**Note:** The proofs for cases $\phi = aA(r)w\phi$, $\phi = \text{blk}(i)\phi$ and $\phi = \text{ff}$ are analogous to the proof for case $\phi = sA(r)w\phi$.

**Case** $\phi = \text{rel}(r)\phi$: From our rule premises we know

\[
\text{excl}(\text{rel}(r), \phi, \psi) = (r, r, \psi) \tag{C.665}
\]

\[
\text{excl}(\varepsilon, r, \psi) \tag{C.666}
\]

By defn of excl (Def. 5.1.2) we know that (C.665) can only be true in the following two subcases (because otherwise, excl would have returned $\bot$).

**Subcase** $\phi = \text{tt}$: Since $\phi = \text{tt}$, by (C.665) and defn of excl (Def. 5.1.2) we know

\[
\text{excl}(\text{rel}(r)\text{tt}, \psi) = (r, r, \psi) \tag{C.667}
\]

By defn of substitution we know

\[
\text{excl}(\text{rel}(r)\text{tt}) = (\varepsilon, r, \psi) \tag{C.668}
\]
C. Proofs for Auxiliary Lemmas

By (C.667) and (C.668) we deduce

\[ \text{excl}(\text{rel}(r)\text{tt})[\text{clr}(X)\text{max} X.\varphi'/X], \psi) = (r, r_\psi) \quad (C.669) \]

\[ \therefore \text{Subcase holds by (C.669).} \]

**Subcase** \( \psi = \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n \): By defn of excl (Def. 5.1.2) and since \( \psi = \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n \) we know that no matter what \( \varphi \) is, the excl function always returns \( \varepsilon \) as the release set of \( \varphi \). Hence by (C.665) and (C.666) we know

\[ \text{excl}(\text{rel}(r) \varphi, \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n) = \text{excl}(\text{rel}(r)\varphi[\text{clr}(X)\text{max} X.\varphi'/X], \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n) = (\varepsilon, r_\varphi) \quad (C.670) \]

\[ \therefore \text{Subcase holds by (C.670).} \]

**Case** \( \varphi = \text{if } c \text{ then } \varphi \text{ else } \psi \): From our rule premises we know

\[ \text{excl}(\text{if } c \text{ then } \varphi \text{ else } \psi, \psi) = (r_\varphi, r_\psi) \quad (C.671) \]

\[ \text{excl}(\text{if } c \text{ then } \varphi \text{ else } \psi, \psi) = (r_\varphi, r_\psi) \quad (C.672) \]

We must consider the following two subcases:

**Subcase** \( \psi \neq \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n \): By (C.671) and defn of excl (Def. 5.1.2) we know

\[ \text{excl}(\varphi_1, \psi) = (r_\varphi, r_\psi) \quad (C.673) \]

\[ \text{excl}(\varphi_2, \psi) = (r_\varphi, r_\psi) \quad (C.674) \]

By (C.672), (C.673) and IH we know

\[ \text{excl}(\varphi_1[\text{clr}(X)\text{max} X.\varphi'/X], \psi) = (r_\varphi, r_\psi) \quad (C.675) \]

By (C.672), (C.674) and IH we know

\[ \text{excl}(\varphi_2[\text{clr}(X)\text{max} X.\varphi'/X], \psi) = (r_\varphi, r_\psi) \quad (C.676) \]

By (C.672), (C.674) and defn of excl (Def. 5.1.2) we can conclude

\[ \text{excl}(\text{if } c \text{ then } (\varphi_1[\text{clr}(X)\text{max} X.\varphi'/X]) \text{ else } (\varphi_2[\text{clr}(X)\text{max} X.\varphi'/X]), \psi) = (r_\varphi, r_\psi) \quad (C.677) \]

By (C.677) and substitution we know

\[ \text{excl}(\text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2[\text{clr}(X)\text{max} X.\varphi'/X], \psi) = (r_\varphi, r_\psi) \quad (C.678) \]

\[ \therefore \text{Subcase holds by (C.678).} \]

**Subcase** \( \psi = \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n \): By defn of excl (Def. 5.1.2) and since \( \psi = \&^k_{\pi_\text{rel}}(\text{rel}(r)\pi)_n \) we know that no matter what \( \varphi \) is, the excl function always returns \( \varepsilon \) as the release set of \( \varphi \).
C. Proofs for Auxiliary Lemmas

Hence by (C.671) and (C.672) we know

\[
\text{excl}(\text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2, \&_{\text{rel}}(\text{rel}(r)_{tt})_n) = \\
\text{excl}(\text{if } c \text{ then } \varphi_1 \text{ else } \varphi_2)[(\text{clr}(X)\max X \varphi')/X], \&_{\text{rel}}(\text{rel}(r)_{tt})_n) = (\varepsilon, r_\varphi) \tag{C.679}
\]

\[\therefore\text{ Subcase holds by (C.679).}\]

Case \(\varphi = \text{tt}\): From our rule premises we know

\[
\text{excl}(\text{tt}, \psi) = (r_\varphi, r_\psi) \tag{C.680}
\]

\[
[(\text{clr}(X)\max X \varphi')/X] \tag{C.681}
\]

From (C.680) and since \(\text{tt} \equiv \text{rel}(\varepsilon) \text{tt}\) we can deduce

\[
\text{excl}(\text{rel}(\varepsilon) \text{tt}, \psi) = (r_\varphi, r_\psi) \tag{C.682}
\]

By (C.681), (C.682) and IH we know

\[
\text{excl}((\text{rel}(\varepsilon) \text{tt})[(\text{clr}(X)\max X \varphi')/X], \psi) = (r_\varphi, r_\psi) \\
\equiv \text{excl}(\text{tt}[(\text{clr}(X)\max X \varphi')/X], \psi) = (r_\varphi, r_\psi) \tag{C.683}
\]

\[\therefore \text{ Case holds by (C.683).}\]

Case \(\varphi = \text{Y}\): From our rule premises we know

\[
\text{excl}(\text{Y}, \psi) = (r_\varphi, r_\psi) \tag{C.684}
\]

\[
[(\text{clr}(X)\max X \varphi')/X] \tag{C.685}
\]

By defn of excl (Def. 5.1.2) we know that (C.684) can only be true if \(\psi = \&_{\text{rel}}(\text{rel}(r)_{tt})_n\) (because otherwise, excl would have returned \(\bot\)). Hence we know

\[
\psi = \&_{\text{rel}}(\text{rel}(r)_{tt})_n \tag{C.686}
\]

Since we know (C.686), then we also know that no matter what \(\varphi\) is (i.e., irrespective of whether \(\text{Y} = \text{X}\) or not), the excl function always returns \(\varepsilon\) as the release set of \(\varphi\). Hence by (C.684), (C.685) and defn of excl (Def. 5.1.2) we can deduce

\[
\text{excl}(\text{Y}, \&_{\text{rel}}(\text{rel}(r)_{tt})_n) = \\
\text{excl}(\text{Y}[(\text{clr}(X)\max X \varphi')/X], \&_{\text{rel}}(\text{rel}(r)_{tt})_n) = (\varepsilon, r_\varphi) \tag{C.687}
\]

\[\therefore \text{ Case holds by (C.687).}\]
References

[1] Akka website. http://www.akka.io.

[2] An architectural blueprint for autonomic computing. Technical report, IBM, 2005.

[3] L. Aceto and A. Ingólfsdóttir. Testing Hennessy-Milner logic with recursion. In FoSSaCS, volume 1578 of LNCS, pages 41–55. Springer, 1999.

[4] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA, 1986.

[5] G. A. Agha, P. Thati, and R. Ziaeii. Formal methods for distributed processing. chapter Actors: A Model for Reasoning About Open Distributed Systems, pages 155–176. Cambridge University Press, New York, NY, USA, 2001.

[6] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. IEEE Trans. Software Eng., 29:634–648, 2003.

[7] J. Armstrong. Programming Erlang. The Pragmatic Bookshelf, 2007.

[8] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In FM, volume 7436 of LNCS, pages 68–84. Springer, 2012.

[9] A. Bauer and Y. Falcone. Decentralised ltl monitoring. In Formal Methods (FM), volume 7436 of LNCS, pages 85–100. Springer, 2012.

[10] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol., 20:14:1–14:64, 2011.

[11] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE ’10, pages 5–14, New York. ACM.

[12] E. Bodden, F. Chen, and G. Rosu. Dependent advice: A general approach to optimizing history-based aspects. In Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD ’09, pages 3–14, New York, NY, USA, 2009. ACM.
[13] E. Bodden and P. Lam. Clara: Partially evaluating runtime monitors at compile time. In *Runtime Verification*, volume 6418 of *LNCS*, pages 74–88. Springer Berlin Heidelberg, 2010.

[14] R. Carlsson. An introduction to core erlang. In *PLI01 (Erlang Workshop)*, 2001.

[15] I. Cassar, A. Francalanza, and S. Said. Improving runtime overheads for detecter. In FESCA, London, UK, 2015, volume 178 of *EPTCS*, pages 1–8, 2015.

[16] I. Castellani, M. Dezani-Ciancaglini, and J. A. Pérez. Self-adaptation and secure information flow in multiparty structured communications: A unified perspective. In *BEAT*, pages 9–18, 2014.

[17] F. Cesarini and S. Thompson. *ERLANG Programming*. O’Reilly, 1st edition, 2009.

[18] F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS’05*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

[19] F. Chen and G. Rosu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588. ACM press, 2007.

[20] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[21] C. Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master’s thesis, University of Malta, 2008. MSc(ICT) Thesis.

[22] C. Colombo. *Runtime Verification and Compensations*. PhD thesis, University Of Malta, 2013.

[23] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.

[24] C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *SEFM*, pages 218–232, 2012.

[25] C. Colombo, G. Pace, and G. Schneider. Larva — Safer monitoring of real-time java programs (Tool paper). In *SEFM*, pages 33–37, 2009.

[26] C. Colombo and G. J. Pace. Fast-forward runtime monitoring - an industrial case study. In *RV*, volume 7687 of *LNCS*, pages 214–228. Springer, 2012.

[27] C. Colombo, G. J. Pace, and P. Abela. Safer asynchronous runtime monitoring using compensations. *Form. Methods Syst. Des.*, 41(3):269–294, 2012.

[28] X. Corporation and P. A. R. Center. The aspectj programming guide, 2012. http://eclipse.org/aspectj/doc/next/progguide/index.html.
References

[29] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[30] M. d’Amorim and K. Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[31] B. D’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and M. Z. LOLA: Runtime Monitoring of Synchronous Systems. In *TIME*, pages 166–174. IEEE, 2005.

[32] N. Decker, M. Leucker, and D. Thoma. jUnitRV - adding runtime verification to jUnit. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 459–464. Springer-Verlag Berlin Heidelberg, Springer, 2013.

[33] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.

[34] D. Drusinsky. Monitoring temporal rules combined with time series. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 114–117. Springer, 2003.

[35] A. Erdödi. Exago: Property monitoring via log file analysis. (Presented at the Erlang User Group Meeting, London, 2010).

[36] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[37] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *SEFM*, volume 7041 of *LNCS*, pages 204–220. Springer, 2011.

[38] A. Francalanza, A. Gauci, and G. J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.

[39] A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *Formal Methods in System Design (FMSD)*, pages 1–36, 2014.

[40] L.-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[41] C. Galea Valletta. Augmenting synchronous and asynchronous monitors in elarva. Final Year Project, 2013.

[42] H. Goldsby, B. Cheng, and J. Zhang. Amoeba-rt: Run-time verification of adaptive software. In *Models in Software Engineering*, volume 5002 of *LNCS*, pages 212–224. Springer Berlin Heidelberg, 2008.

[43] P. Haller and F. Sommers. *Actors in Scala*. Artima Inc., USA, 2012.
References

[44] K. Havelund. A scala dsl for rete-based runtime verification. In *Runtime Verification*, volume 8174 of *LNCS*, pages 322–327. Springer Berlin Heidelberg, 2013.

[45] K. Havelund and G. Roșu. Monitoring java programs with java pathexplorer. In *RV (in connection with CAV)*, volume 55:2 of *ENTCS*, pages 200–217, 2001.

[46] A. Hernandez. Yaws 1.89: Directory traversal vulnerability. Online at http://www.exploit-db.com/exploits/15371/, 2010. Accessed on 1/12/2014.

[47] A. Holovaty and J. Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right, Second Edition*. Apress, Berkely, CA, USA, 2nd edition, 2009.

[48] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. SEAMS ’08, pages 97–104. ACM, 2008.

[49] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar. Building user-defined runtime adaptation routines for stream processing applications. *Proc. VLDB Endow.*, 5(12):1826–1837, 2012.

[50] M. A. Kalareh. *Evolving Software Systems for Self-Adaptation*. PhD thesis, University of Waterloo, Ontario, Canada, 2012.

[51] S. Kell. A survey of practical software adaptation techniques. *J. UCS*, 14(13):2110–2157, 2008.

[52] Z. Kessin. *Building Web Applications with Erlang: Working with REST and Web Sockets on Yaws*. O’Reilly Media, 2012.

[53] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A runtime assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[54] A. Krasnopolski. AOP for Erlang. http://erlaop.sourceforge.net/.

[55] M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293 – 303, 2009.

[56] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 4(1-2):2–16, 2005.

[57] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Computer Security ESORICS 2010*, volume 6345 of *LNCS*, pages 87–100. Springer Berlin Heidelberg, 2010.

[58] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning, 2011.

[59] F. Long, V. Ganesh, M. Carbin, S. Sidiрогlou, and M. Rinard. Automatic input rectification. *ICSE*, pages 80–90. IEEE Press, 2012.
[60] F. Long, S. Sidiropoglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. *SIGPLAN Not.*, 49:227–238.

[61] P. O. Meredith and G. Rosu. Runtime verification with the rv system. In *RV*, volume 6418 of *LNCS*, pages 136–152. Springer, 2010.

[62] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[63] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: Framework, approaches, and styles. *ICSE Companion ’08*, pages 899–910, New York, NY, USA, 2008. ACM.

[64] B. C. Pierce, editor. *Advanced Topics in Types and Prog. Languages*. MIT Press, 2005.

[65] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee. Enhancing Availability & Security through Failure-oblivious Computing. *OSDI*, pages 303–316. USENIX, 2004.

[66] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.

[67] S. Saudrais, A. Staikopoulos, and S. Clarke. Using specification models for runtime adaptations.

[68] K. Sen, A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pages 418–427, 2004.

[69] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment, Second Edition*. Addison-Wesley Professional, 2nd edition, 2008.

[70] H. Svensson, L.-Å. Fredlund, and C. Benac Earle. A unified semantics for future erlang. In *Erlang Workshop*, pages 23–32. ACM, 2010.

[71] C. Talcott, M. Sirjani, and S. Ren. Comparing three coordination models: Reo, arc, and rrd. *ENTCS*, 194(4):39–55, 2008.

[72] B. Venneri, M. Coppo, and M. Dezani-Ciancaglini. Self-adaptive monitors for multiparty sessions. *Euromicro*, pages 688–696, 2014.

[73] S. Vinoski. Yaws: Yet another web server. *IEEE Internet Computing*, 15(4):90–94, 2011.

[74] J. Zhang and B. H. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006.