GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Doctoral Thesis

A choreographic approach to model-driven testing of message-passing systems

PhD Program in Computer Science: XIV cycle

Supervisors:

Prof. Roberto Guanciale
roberto.guanciale@kth.se

Prof. Emilio Tuosto
emilio.tuosto@gssi.it

Author:

Alex Coto
alex.coto@gssi.it

December, 2023

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Abstract

As choreographic development garners interest for enabling open, scalable applications, two main approaches have been utilized: top-down and bottom-up development. The former involves generating code from global specifications, manually or automatically, while the latter analyzes the composition of existing components by deriving a global specification encapsulating their coordination protocols. Such approaches assure certain software quality measures; for instance, if global specifications meet certain conditions, deadlock freedom can be guaranteed.

Despite these assurances, software testing remains pivotal, especially when human intervention is required for implementing local computations or maintaining and evolving software. In this light, we advocate for the utilization of choreographic models, both global and local, as a foundation for model-driven testing of message-passing applications. To our knowledge, this application domain lacks a systematic approach to testing.

Our research introduces a new approach to test synthesis based on choreographic models, focusing on an abstract framework that bridges the gap between these models and executable tests. This framework aids in identifying potential defects resulting from dynamic changes in components and in determining if local computation bugs may breach causal dependencies in globally specified communications. The framework's black-box testing approach, considering that the components' code might not be available, ensures widespread applicability.

Additionally, we implement a toolchain that synthesizes executable tests from abstract choreographic models, utilizing local session types to represent abstract local protocols. These protocols are designed to be statically verifiable, ensuring generated tests are free from type errors. Our toolchain also generates specific Python tests, effectively transforming abstract models into concrete tests.

Among our contributions are defining key concepts such as test cases, oracle, test compliance, and suitability, and introducing a library for defining local multiparty session types in Python, supported by a prototype implementation for generating executable tests. Our approach to testing, based on choreographic models, addresses some of the challenges currently plaguing testing practices in message-passing applications, thus marking a stride forward in efficient testing and verification of distributed systems.

Contents

List of Figures

Chapter 1

Introduction

Computing systems, undeniably integral to modern life, underpin tasks ranging from the mundane, such as posting a picture on a social network, to life-critical applications like remotely operated surgical robots or insulin pump management. As we grow increasingly reliant on these systems, the need for rigorous design, analysis, and verification mechanisms becomes paramount. This is underscored by the awareness that unreliable systems can lead to considerable economic loss, or even, in extreme cases, loss of life.

For example, in 2018 and 2019, two crashes involving the Boeing 737 Max aircraft led to the deaths of 346 people. Investigations revealed that a system known as the Maneuvering Characteristics Augmentation System (MCAS) was a major factor in both crashes. The MCAS was designed to automatically adjust the aircraft's nose position to prevent stalling, but incorrect data from a single faulty sensor could cause the system to push the nose down repeatedly, leading to a dive the pilots couldn't recover from [1].

The Therac-25 incidents were are another instance that cost multiple lives. Between 1985 and 1987, a series of radiation overdoses from the Therac-25 radiation therapy machine led to at least five deaths. The machine had a software bug that, under certain conditions, allowed it to administer radiation doses that were hundreds of times greater than normal. The lack of hardware interlocks (which would have made such an overdose impossible) was a major factor in the accidents [2, 3].

A significant portion of research in computer science has been directed towards developing new mechanisms to facilitate the design and verification of computing systems. Two prevalent approaches in this domain are formal methods and testing [4, 5]. Formal methods employ precise mathematical and logic techniques for the specification, development, and verification of systems, while testing typically involves executing the system, or its components, with specific inputs and validating the produced outputs [6].

Formal methods, although popular especially in academia, can be complex to apply, demanding considerable expertise. They are typically performed on simplified models of the system, which may not encompass all aspects of the actual implementation [7]. On the other hand, testing operates directly on the system's implementation, capturing issues that may arise from the implementation details or the runtime environment. However, testing arguably delays the discovery of defects until the system is implemented and/or executed, and it can only cover a finite set of inputs and execution paths, providing less comprehensive assurance than formal methods [8].

Distributed systems, powering a vast array of modern applications from cloud services to e-commerce platforms, present unique challenges that amplify the importance of rigorous testing. The concurrent and asynchronous nature of these systems can result in unpredictable behaviours, including race conditions and synchronization issues. Testing can help isolate these potential problems and ensure the system functions as expected. Furthermore, testing can validate the system's resilience to network issues, such as latency, packet loss, and network partitions, which are common challenges in distributed systems [9]. Considering the security implications, rigorous validation is also imperative to identify potential vulnerabilities and enhance system resilience against various forms of attacks.

Model-based testing employs models of the system's behaviour to automatically generate test cases. This approach can increase test coverage[1] and decrease the effort required to create and maintain test cases[2]. Furthermore, it helps ensure that the tests remain consistent with the system's specification, as reflected in the model [10]. Another advantage of model-based testing is that it can report failures consistently. For instance, finding the "shortest" test sequence to a specific failure often enables the user to inspect not only the code of that test, but also the corresponding abstract test from which it was generated [10].

Choreography models, originally proposed by the W3C in the WS-CDL specification [11], provide an effective way to specify and visualize the intended communication behaviour of a distributed system. They capture the global view of the interactions among multiple participants in a distributed system, with each participant represented as a role in the choreography. If these models satisfy certain conditions, it becomes possible to ascertain some properties about them that guarantee that the systems being modelled will behave reliably (i.e., they will not deadlock, etc.).

[1] e.g., by supporting systematic coverage of the system-under-test's functionality [10]
[2] e.g., by automatically re-generating test cases when changes are introduced to the model [10]

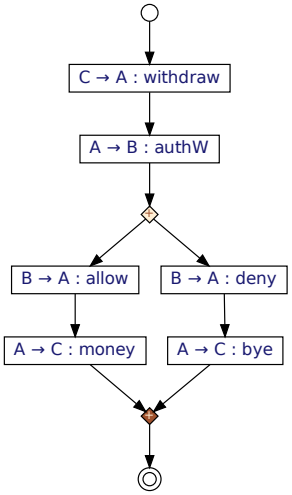


FIGURE 1.1: A (very) simplified global view of an ATM protocol.

Choreographies provide an overview of all interactions among the different parts of the system, serving as a blueprint for the expected behaviour. They ensure that all participants in the system adhere to the agreed-upon interaction protocol, thus reducing the potential for communication mismatches and subsequent failures. The typical workflow intends to achieve the so-called *correctness-by-construction* principle [11]. For instance, the diagram in Figure 1.1 is an example of a global view for a simplified protocol, representing the main interactions of a client willing to withdraw some cash from an ATM.

The protocol starts with the *interaction* C→A: `withdraw`, where the client C communicates to the ATM A their intention to withdraw cash. This interaction triggers the next one A→B: `authW` in which A asks the client's bank B to authorize the withdrawal. A distributed choice starts at the *branching* point ◈, where the bank B decides whether to deny or grant the withdrawal. Note that the choice is non-deterministic due to the abstraction level of this model (see discussion below). Depending on the local decision of B, the next interaction is either B→A: `deny` or B→A: `allow`. In each case the client is notified of the decision with interactions A→C: `bye` (in the first case) or A→C: `money` if the operation is granted by the bank.

Choreographic approaches are gaining momentum, and have proven themselves to be valuable approaches in both academy and industry (e.g., [12–23]). The correctness-by-construction principle of choreographic models is usually realized through the identification of *well-formedness* conditions on global views. These are sufficient conditions that guarantee that the protocol can be executed distributively, without breaking consistency between the global state and the local knowledge of participants. In particular, formal choreographic approaches (such as [20–29] to mention but a few) study notions of

well-formedness to guarantee the safety of communications (usually, deadlock-freedom, no message losses, etc.). The local view of a protocol indeed provides a specification for the behaviour of each component "in isolation". In this way, the local view yields a set of computational units enacting the protocol specified by the global view. For instance, the local view of the bank B above consists of an artefact waiting for an `authW` message from A to which it replies by sending either a `deny` or an `allow` message.

1.1 Motivation

A main source of problems in distributed protocols is that the knowledge about the global state of the computation is scattered across the various participants involved in the execution. This implies that a participant may not be able to infer what is the "global" state of the execution. For instance, in the example above, the bank B is not aware of the fact that the client sent a request of withdrawal until it receives the request of authorization from A. Such uncertainty makes it problematic to reach consensus among participants when distributed choices must be taken. And, for the protocol to run "smoothly", the partial knowledge of each participant should be consistent with respect to the global state of the protocol. For distributed choices, this boils down to requiring *awareness* from each participant about which branch to follow. For instance, in the example above, the bank is aware of the choice since it decides what to do next, and the other participants can discriminate the bank's choice from the messages they exchange throughout the execution.

Even in a correctly implemented choreographic solution problems may arise. We list three major causes of possible disruption:

Local computation As recalled above, formal choreographies focus on the interactions among components while abstracting away from local computations. Therefore, errors may still be introduced when developing code. For instance, a component expected to receive an integer and return a string, after inputting the integer may diverge on a local computation before delivering the expected string and cause a malfunction in the communication protocol.

Evolution For example, to reduce the communication overhead, a component may be modified so that two outputs are merged into one to spare an interaction. Besides introducing bugs in the new code for local computations, these changes may alter the original design.

Openness Applications are increasingly being constructed by integrating independent computational components that are readily available off-the-shelf, and, in many

cases, developers may not have any control over these elements. This method is particularly prevalent in the development of service-oriented architectures. New releases or modifications of third-party components (libraries, run-time support, etc.) may introduce malfunctions in applications using it. For example, a new release of a service invoked by an application may enrich the spectrum of possible messages delivered to some components not designed to handle such new messages.

Depending on the semantics used, choreographic models can precisely specify the order of messages, which is essential for systems where the sequence of interactions matters. This level of precision is useful when testing, as it allows the developer to identify deviations from the expected behaviour. As the system evolves, a choreography specification can be updated to reflect new or changed interactions. The updated choreography can then be used to generate new tests, ensuring that testing keeps pace with system development. Choreographic models, in their formal representation of interactions among system components, lay out the expected behaviour and order of message exchanges for each participant, bringing much-needed clarity to complex distributed protocols. This precision becomes indispensable in testing, as it assists in pinpointing deviations from the predicted system behaviour. Furthermore, as systems evolve and interactions transform, choreographies can be updated to mirror these changes, ensuring that testing remains in lockstep with system development. The ability to derive executable tests from these choreographies further augments their value by streamlining the validation process, ensuring system components adhere to the prescribed communication protocols. These characteristics therefore position choreographies as a potent tool in mitigating the challenges posed by local computation errors, evolution of system components, and the uncertainty associated with incorporating independently developed, off-the-shelf components. These characteristics make them particularly interesting for testing message-passing systems.

Our aim is therefore to complement the correctness-by-construction principle of choreographies with model-driven testing. We argue that model-based testing can help to find errors arising from the aforementioned causes. We argue that, by synthesizing executable tests from choreographies, developers can automate the process of validating the adherence of system components to the intended communication protocol.

1.2 Contributions

At the core of our contribution is the development of an abstract framework for testing message passing systems. This abstract framework captures the essential components of formal choreographic models and is integral to our test generation algorithm. One of

our primary contributions is our formal framework, encompassing the definition of test cases, oracle, test compliance, and suitability. Moreover, we provide a toolchain that implements an abstract test generation algorithm based on this framework. Both the framework and the toolchain are designed to be extensible, allowing for the incorporation of new test generation algorithms and oracles. This abstract framework holds the capacity to check if local computation bugs may lead to violations of causal dependencies in globally specified communications. These elements were originally presented in [30, 31], through multiple use cases (of which we include a few in this thesis), and further refined in [32] and Chapter 3.

Notably, the utility of our framework extends to identifying defects resulting from component changes due to evolution or dynamic alterations. Such modifications could potentially disrupt the communication interface, making it inconsistent with globally specified components. Considering the possibility that the code for new or updated components might be inaccessible in certain contexts, our framework has been developed under the assumption that the components' code may not be available. This assumption is particularly relevant in the context of service-oriented architectures, where components are often developed independently and are available off-the-shelf.

To enhance the practicality of our work, we have also extended our toolchain with features that allow to synthesize executable tests based on our abstract choreographic models. To achieve this, the contributions can be narrowed down to three main elements. Firstly, we develop a local session types library that allows for communication protocols to be represented as generic types in the Python programming language. These local session types have been designed to be statically verifiable by an off-the-shelf Python type checker[3], providing an additional layer of validation. Secondly, using these local session types, we develop a mechanism to derive executable Python tests from choreographic models. Finally, to support this process, we also provide an extension providing editor support for syntax highlighting, autocompletion, graphical feedback and additional functionalities when working with our choreographic models, providing a more user-friendly interface for developers. These elements are presented and discussed in Chapter 4.

1.3 Outline

The remainder of the thesis is structured as follows.

[3]Such as Pyright [33], to which we also contributed with bug reports that arose during implementation.

In Chapter 2 we present the required background and foundations for our work. We give a formal description of the models we use: Communicating Finite State Machines, Labelled Transition Systems, and g-choreographies. We end the chapter with a discussion of related work.

Chapter 3 deals with the formalization of our framework. The algorithm for test generation is presented here, using both a formal description and illustrative examples. We also discuss ChorTest , a toolchain implementation of our framework: its technicalities, usage instructions, and also walk through some examples. Finally, we describe the mutation testing approach we use to validate our approach, introducing a set of mutation operators and discussing the obtained results.

Chapter 4 introduces a library for defining local multiparty session types in Python, and a mechanism to generate executable tests derived from the choreographic models. We discuss the technicalities of the implementation, and also present some examples and results.

Chapter 5 is a final discussion, encompassing the issues we've found along the research, possible alternatives, and future directions.

1.4 List of Publications

- Alex Coto, Roberto Guanciale, and Emilio Tuosto. On testing message-passing components. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2020. doi: 10.1007/978-3-030-61362-4_2

- Alex Coto, Roberto Guanciale, and Emilio Tuosto. Choreographic development of message-passing applications - A tutorial. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2020. doi: 10.1007/978-3-030-50029-0_2

- Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.*, 123:100712, 2021. doi: 10.1016/j.jlamp.2021.100712

- Alex Coto, Franco Barbanera, Ivan Lanese, Davide Rossi, and Emilio Tuosto. On formal choreographic modelling: A case study in EU business processes. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, volume 13701 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2022. doi: 10.1007/978-3-031-19849-6_13

Chapter 2

Background

Computing systems, at a high level of abstraction, can be considered as black boxes that map inputs to their appropriate output values. These systems can be described by how they transform an initial state to a final state.

In this classic view of computing systems, non-termination is envisaged as a catastrophic outcome. This view is not suitable to describe some of the systems we use nowadays, for which non-termination is an actual desirable property: highly diverse systems, including (but not limited to) web servers, communication protocols, operating systems, embedded controllers and many more [35]. The term *reactive systems* denotes these software and hardware systems that exhibit a (usually) non-terminating behaviour, interacting through observable events.

Due to the inherent complexity of reactive systems, many attempts have been made to design sophisticated validation methods. To ensure that reactive systems meet their requirements, two of the most common approaches are *testing* [36] and *formal verification* [35]. Among the latter, *session types* have recently emerged as the base for new programming languages and software development methods for communication-intensive distributed systems [37].

The first section in this chapter introduces the formal models relevant to understanding our work: Multiparty Session Types (MPST) and choreography models, Communicating Finite State Machines (CFSM), and the projection operations that allow us to transition between global and local models. As a starting point, we explore *session types*, a type-based approach for specifying and verifying communication protocols in concurrent systems. Building on the foundational concept of session types, we then introduce *g-choreography* models. These models extend session types to scenarios involving multiple interacting parties, improving expressiveness and providing a more holistic view

of protocols. Next, we discuss *communicating finite state machines* (CFSMs), a formal model for concurrent systems that captures the dynamics of message-passing interactions between parallel processes. Finally, we examine the *projection operations* that enable a transition from global choreography models to local session types. These operations allow developers to automatically derive the local communication protocols for individual participants from the global specification, ensuring consistency and correctness across the system.

The second section of this chapter gives a brief overview of software testing and, more specifically, of the *model-based testing* (MBT) paradigm. We end the chapter with a discussion of related work, focusing on the use of behavioural types models in the context of verifying message passing systems.

2.1 Session Types

One of the many formalisms proposed to structure interaction and reason about communicating processes and their behaviour are *session types*. Through them, the conformance of programs to protocols can be checked. The structure of a conversation can be abstracted as a type through a specific syntax, which is then used to validate programs [38]. These types have their roots in process calculi, specifically in typed π-calculus. The core idea behind session types is that we describe communication protocols as types, in a way that they can be checked either statically (at compile time) or dynamically (at runtime). Interest on session types implementations has arisen recently, leading to session types being used across many programming languages (see, e.g., [39–41]).

To give an example of how a protocol can be described as a session type, we show a (very) simplified version of the Simple Mail Transfer Protocol (SMTP) protocol [42] in Figure 2.1. The $\oplus$ symbol represents a choice on the client for one of the two branches (EHLO and QUIT). The `!` symbol accounts for the sending of a message, and if the client chooses the EHLO branch, the protocol can be repeated. Otherwise, communication ends after the client has chosen the QUIT branch.

```
SMTPClient = ⊕ {
  EHLO: !Domain.!FromAddress.!ToAddress.!Message.SMTPClient
  QUIT: end
}
```

FIGURE 2.1: A simplified version of the SMTP protocol as a session type

Conversely, the session type for the server is known as the *dual* type, shown in Figure 2.2. Whenever we sent a message, now we wait for one (represented by the `?` symbol). Whenever we made a choice before, now we offer one (through the & symbol).

```
SMTPServer = & {
  EHLO: ?Domain.?FromAddress.?ToAddress.?Message.SMTPServer
  QUIT: end
}
```

FIGURE 2.2: The dual session type of the previous SMTP protocol example

The notion of duality in binary session types (i.e., session types with two participants) refers to the principle that for every action in a session, there is a corresponding opposite action in the other session participant. This means if one party sends a message, the other party must receive that message, and vice versa. In other words, session types describe a protocol between two or more parties, and this protocol's duality ensures that for every output action of one participant, there is a corresponding input action from the other participant. Session types can be formally specified in terms of these dual actions, where each action in a session type has a corresponding dual action in the complementary session type. For example, if a session type A involves sending a message of type m and then continuing as session B, its dual would involve receiving a message of type m and then continuing as the dual of B. These specifications can then be used to verify statically that both the client and a server conform to the protocol.

2.1.1 Multiparty Session Types

Multiparty Session Types (MPST) are a generalization of binary session types to multiple parties [43]. One of the core ideas behind Multiparty Session Types is to represent the allowed interactions in a top-down manner, as a *global type*. These global types can then be projected into *local types*, which describe interactions from the point of view of a particular participant. Communication can then be checked against these local types, either statically (through the compiler) or dynamically (through run-time monitoring).

One of the most common examples when introducing MPST is the "two-buyer protocol", which is intended to be a simplification of a financial protocol [43]. Essentially, two buyers ("Buyer1" and "Buyer2") wish to purchase a book from a bookseller ("Seller"), negotiating the cost between them.

The protocol proceeds as follows:

1. Buyer 1 sends the title of the book to the seller (line 2).
2. The seller sends the price to Buyer 1 and Buyer 2 (line 3).
3. Buyer 1 sends Buyer 2 the amount that Buyer 2 should pay (line 5).
4. Buyer 2 can choose to:

```
1  global protocol TwoBuyer(role Buyer1, role Buyer2, role Seller) {
2    title(String) from Buyer1 to Seller;
3    price(Currency) from Seller to Buyer1, Buyer2;
4    rec loop {
5      share(Currency) from Buyer1 to Buyer2;
6      choice at Buyer2 {
7        accept() from Buyer2 to Buyer1;
8        deliveryAddress(String) from Buyer2 to Seller;
9        deliveryDate(Date) from Seller to Buyer2;
10     } or {
11       reject() from Buyer2 to Buyer1;
12       continue loop;
13     } or {
14       quit() from Buyer2 to Buyer1, Seller;
15     }
16   }
17 }
18
```

FIGURE 2.3: A global protocol in the Scribble language for the two-buyer protocol

(a) Accept the offer, at which point it sends a delivery address to the seller and receives a delivery date (lines 7-9).

(b) Reject the offer, and await another offer from Buyer 1 (lines 11-12).

(c) End the protocol (line 14).

Several languages have been designed to represent these kinds of protocols. *Scribble*, for instance, is a language for describing protocol that is based on the theory of multiparty session types [44]. Fowler [42] gives a representation of the previous protocol in the Scribble language, as seen in Figure 2.3. This global specification can then be projected into local ones. For example, the code corresponding to Buyer2 in the previous example is shown in Figure 2.4.

2.1.2 G-choreographies

From a global point of view, *choreographies* are syntactic descriptions of the overall coordination of a system, in terms of interactions between autonomous principals. A choreography captures how two or more endpoints (or nodes) exchange messages during execution from a global viewpoint, instead of defining individually the behaviour of each endpoint. It assumes no centralization, and uses events and publish/subscribe mechanisms to establish collaboration. Being a visual representation of multiparty session types, their main advantage lies that they are easier to understand and reason about.

Choreographic approaches are particularly suited to model microservice architectures. Microservice-based architectures are based on the principle of composition of small services, isolated in their own processes and communicating via lightweight mechanisms

```
local protocol TwoBuyer
    at Buyer2(role Buyer1, role Buyer2, role Seller) {
  price(Currency) from Seller;
  rec loop {
    share(Currency) from Buyer1;
    choice at Buyer2 {
      accept() to Buyer1;
      deliveryAddress(String) to Seller;
      deliveryDate(Date) from Seller;
    } or {
      reject() to Buyer1;
      continue loop;
    } or {
      quit() to Buyer1, Seller;
    }
  }
}

```

FIGURE 2.4: A local projection of the two-buyer protocol into one of the buyers in the Scribble language

such as message-passing [45]. This approach of making services independent both in their development and deployment has several advantages regarding maintainability, scalability and other aspects, but it also comes with the issue of requiring more complex communication among services involved. Orchestration tends to be more popular (due to its simplicity of use and easier ways to manage complexity), but it can lead to service coupling and an uneven distribution of responsibilities. The culture of decentralization in microservices represents the natural application scenario for choreographies in order to achieve collaboration.

Graphical choreographies (a.k.a. global graphs or g-choreographies, for short) are a graphical representation of global choreographies, introduced by Denièlou and Yoshida [46]. These global graphs feature expressive constructs (forking, merging, joining, etc.) for representing application-level protocols. Lange et al. have proposed an algorithm for automatically deriving a choreography (expressed as a global graph) from a set of CFSMs (Communicating Finite State Machines), given that they satisfy a property known as generalized multiparty compatibility (GMC) [47]. This property also guarantees that the CFSMs interact without deadlocks and other communication errors.

The TwoBuyer protocol from previous sections can be represented through a global graph as pictured in Figure 2.5. The + token indicates a choice (as well as the end of a choice section). Although it was not used in this example, the | token also allows expressing parallel execution. Loops can be expressed using a backward arrow, as usual in other kinds of diagrams.

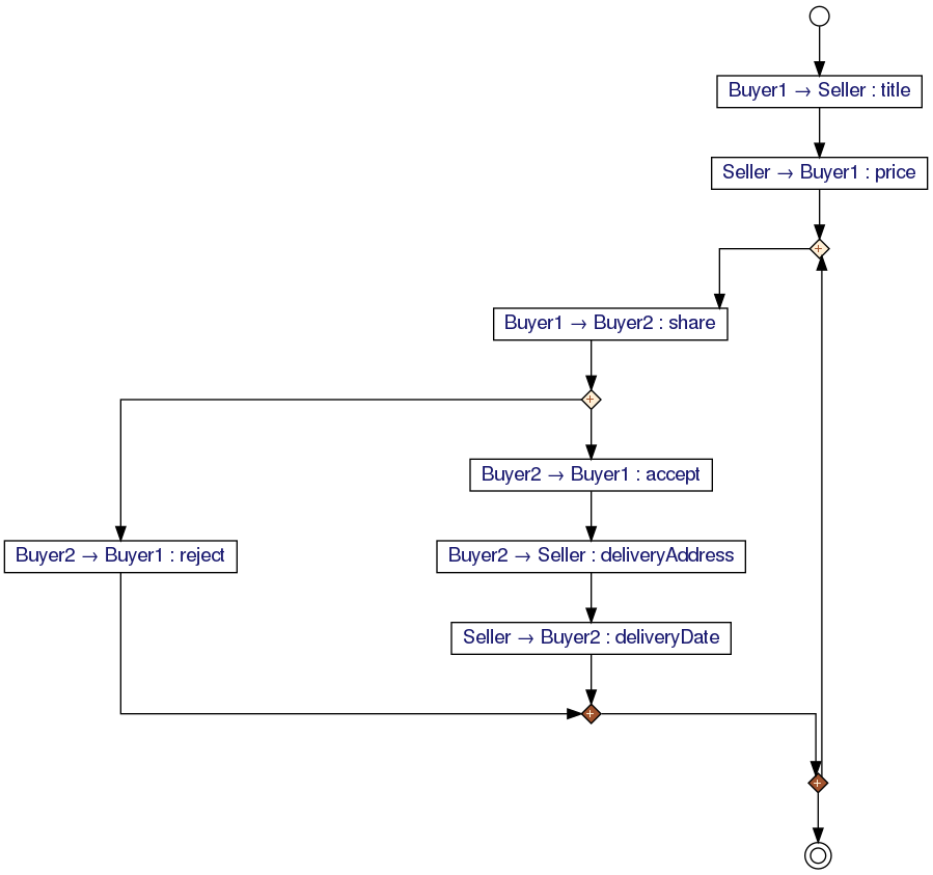


FIGURE 2.5: The two-buyer protocol as a g-choreography

More formally, a g-choreography is a term derivable from the following grammar:

$$
\begin{array}{lcll}
\mathsf{G} & ::= & (\mathtt{o}) & \text{empty} \\
& | & \mathsf{A}{\to}\mathsf{B}\colon \mathtt{m} & \text{interaction} \\
& | & \mathsf{G} \mid \mathsf{G} & \text{fork} \\
& | & \mathsf{G} + \mathsf{G} & \text{choice} \\
& | & \mathsf{G}; \mathsf{G} & \text{sequential} \\
& | & \mathsf{repeat}\ \mathsf{G} & \text{iteration}
\end{array}
$$

The empty choreography $(\mathtt{o})$ yields no interactions; trailing occurrences of $(\mathtt{o})$ may be omitted. An interaction $\mathsf{A}{\to}\mathsf{B}\colon \mathtt{m}$ represents the exchange of a message of type $\mathtt{m}$ between A and B, provided that $\mathsf{A} \neq \mathsf{B}$. We remark that data are abstracted away: in $\mathsf{A}{\to}\mathsf{B}\colon \mathtt{m}$, the message $\mathtt{m}$ is not a value and should rather be thought of as (the name of) a data type[1].

[1]We leave implicit the grammar of data types; in the examples we assume that `m` ranges over basic types such as `int`, `bool`, `string`, etc.

G-choreographies can be composed sequentially or in parallel ($\mathsf{G}; \mathsf{G}'$ and $\mathsf{G} \mid \mathsf{G}'$). A (non-deterministic) choice $\mathsf{G}_1 + \mathsf{G}_2$ specifies the possibility to continue according to either G_1 or G_2. The body G in an iteration $\mathsf{repeat}\ \mathsf{G}$ is repeated until a participant in G (non-deterministically) chooses to exit the loop. Strictly speaking iterative choreographies are not crucial for testing; in fact, it is commonplace in software testing (as well as in other verification techniques such as bounded model-checking) to consider finitary unfoldings of loops. One can think of $\mathsf{repeat}\ \mathsf{G}$ as syntactic sugar for $\underbrace{\mathsf{G}; \cdots; \mathsf{G}}_{n-\text{times}}$.

Note that, for simplicity, our examples only deal with non-iterative choreographies.

The semantics of a g-choreography, as defined in [26, 48], is a family of pomsets (partially ordered multisets); each pomset in the family is the partial order of events occurring on a particular "branch" of the g-choreography. Events are therefore labelled by *(communication) actions* l occurring in the g-choreography. The output of a message $\mathtt{m} \in \mathcal{M}$ from participant $\mathsf{A} \in \mathcal{P}$ to participant $\mathsf{B} \in \mathcal{P}$ is denoted by $\mathsf{A}\,\mathsf{B}!\mathtt{m}$, while the corresponding input is denoted by $\mathsf{A}\,\mathsf{B}?\mathtt{m}$. More formally,

$$\mathcal{L}_{\text{act}} = \{\mathsf{A}\,\mathsf{B}!\mathtt{m}, \mathsf{A}\,\mathsf{B}?\mathtt{m} \mid \mathsf{A}, \mathsf{B} \in \mathcal{P} \text{ and } \mathtt{m} \in \mathcal{M}\}$$

is the set of (communication) *actions* and l ranges over $\mathcal{L}_{\text{act}}$. The subject of an action is defined as $\mathsf{sbj}(\mathsf{A}\,\mathsf{B}!\mathtt{m}) = \mathsf{A}$ and $\mathsf{sbj}(\mathsf{A}\,\mathsf{B}?\mathtt{m}) = \mathsf{B}$.

The semantics $[\![(\mathsf{o})]\!]$ is the set $\{\epsilon\}$ containing the empty pomset ϵ, while for interactions we have

$$[\![\mathsf{A} \to \mathsf{B} : \mathtt{m}]\!] = \left\{ \left[\mathsf{A}\,\mathsf{B}!\mathtt{m} \longrightarrow \mathsf{A}\,\mathsf{B}?\mathtt{m} \right] \right\}$$

namely, the semantics of an interaction is a pomset where the output event precedes the input event. The semantics of the other operations is basically obtained by composing the semantics of sub g-choreographies. More precisely,

- for a choice we essentially have $[\![\mathsf{G} + \mathsf{G}']\!] = [\![\mathsf{G}]\!] \cup [\![\mathsf{G}']\!]$;
- the semantics of the parallel composition $\mathsf{G} \mid \mathsf{G}'$ is built by taking the disjoint union of each pomset in $[\![\mathsf{G}]\!]$ with each one in $[\![\mathsf{G}']\!]$;
- the semantics of the sequential composition $[\![\mathsf{G}; \mathsf{G}']\!]$ is the disjoint union of each pomset in $[\![\mathsf{G}]\!]$ with each one in $[\![\mathsf{G}']\!]$ and, for every participant A, making every output of A in $[\![\mathsf{G}]\!]$ precede all events of A in $[\![\mathsf{G}']\!]$.

Notice that these are *asynchronous* semantics. This notion can be better illustrated through the following example.

Example 2.1. *Consider* $\mathsf{G} = \mathsf{A}{\to}\mathsf{B}\colon \mathtt{x}; \mathsf{A}{\to}\mathsf{B}\colon \mathtt{y}$. *The semantics for this g-choreography would be:*

$$[\![\mathsf{G}]\!] = \left\{ \left[\begin{array}{ccc} \mathsf{A}\,\mathsf{B}!x & \longrightarrow & \mathsf{A}\,\mathsf{B}?x \\ \downarrow & & \downarrow \\ \mathsf{A}\,\mathsf{B}!y & \longrightarrow & \mathsf{A}\,\mathsf{B}?y \end{array} \right] \right\}$$

Note that there is no causal relationship between the events $\mathsf{A}\,\mathsf{B}?x$ *and* $\mathsf{A}\,\mathsf{B}!y$.

Example 2.2. *Consider* G_{ATM} *from Figure 1.1. We have*

$$[\![\mathsf{G}_{ATM}]\!] = \left\{ \begin{array}{l} \left[\begin{array}{ccccccc} \mathsf{C}\,\mathsf{A}!withdraw & \longrightarrow & \mathsf{C}\,\mathsf{A}?withdraw & \longrightarrow & \mathsf{A}\,\mathsf{B}!authW & \longrightarrow & \mathsf{A}\,\mathsf{B}?authW \\ & & & & & & \downarrow \\ & & \mathsf{A}\,\mathsf{C}?bye & \longleftarrow & \mathsf{A}\,\mathsf{C}!bye & \longleftarrow & \mathsf{B}\,\mathsf{A}?deny \longleftarrow \mathsf{B}\,\mathsf{A}!deny \end{array} \right], \\ \left[\begin{array}{ccccccc} \mathsf{C}\,\mathsf{A}!withdraw & \longrightarrow & \mathsf{C}\,\mathsf{A}?withdraw & \longrightarrow & \mathsf{A}\,\mathsf{B}!authW & \longrightarrow & \mathsf{A}\,\mathsf{B}?authW \\ & & & & & & \downarrow \\ & & \mathsf{A}\,\mathsf{C}?money & \longleftarrow & \mathsf{A}\,\mathsf{C}!money & \longleftarrow & \mathsf{B}\,\mathsf{A}?allow \longleftarrow \mathsf{B}\,\mathsf{A}!allow \end{array} \right] \end{array} \right\}$$

For the sake of illustration, the singleton

$$\left\{ \left[\begin{array}{ccccccc} & & & & & & \mathsf{A}\,\mathsf{C}?bye \longleftarrow \mathsf{A}\,\mathsf{C}!bye \longleftarrow \mathsf{B}\,\mathsf{A}?deny \longleftarrow \mathsf{B}\,\mathsf{A}!deny \\ & & & & & & \uparrow \\ \mathsf{C}\,\mathsf{A}!withdraw & \longrightarrow & \mathsf{C}\,\mathsf{A}?withdraw & \longrightarrow & \mathsf{A}\,\mathsf{B}!authW & \longrightarrow & \mathsf{A}\,\mathsf{B}?authW \\ & & & & & & \downarrow \\ & & \mathsf{A}\,\mathsf{C}?money & \longleftarrow & \mathsf{A}\,\mathsf{C}!money & \longleftarrow & \mathsf{B}\,\mathsf{A}?allow \longleftarrow \mathsf{B}\,\mathsf{A}!allow \end{array} \right] \right\}$$

is the semantics of the g-choreography obtained by replacing choice with parallel composition in G_{ATM}.

The language of a g-choreography G, written $\mathcal{L}[\mathsf{G}]$, is the closure under prefix of the set of all *linearizations* of $[\![\mathsf{G}]\!]$, where a linearization of a pomset is a permutation of its events that preserves the order of the pomset.

Example 2.3. *The language of the last pomset in Example 2.2 is the set of prefixes of words obtained by concatenating*

$$\mathsf{C}\,\mathsf{A}!withdraw\ \mathsf{C}\,\mathsf{A}?withdraw\ \mathsf{A}\,\mathsf{B}!authW\ \mathsf{A}\,\mathsf{B}?authW$$

with both

$$\mathsf{B}\,\mathsf{A}!deny\ \mathsf{B}\,\mathsf{A}?deny\ \mathsf{A}\,\mathsf{C}!bye\ \mathsf{A}\,\mathsf{C}?bye\ ,\ \mathsf{B}\,\mathsf{A}!allow\ \mathsf{B}\,\mathsf{A}?allow\ \mathsf{A}\,\mathsf{C}!money\ \mathsf{A}\,\mathsf{C}?money$$

For a more detailed presentation of these semantics, the interested reader can consult [26, 48].

2.1.3 Communicating Finite State Machines

Communicating Finite State Machines (CFSMs) can be seen as an automata-theoretic presentation of Milner's CCS[2]. They were proposed in the '80s to represent and analyse communication protocols from a local point of view. Brand and Zafiropulo proposed them to address the problem of ensuring certain generally desirable properties, thus proving the "well-formedness" of certain protocols [49].

As their name implies, CFSMs are based on finite-state machines. The intuition behind them lies in the fact that a single finite-state machine cannot describe all possible protocols, due to (for example) its inability to describe protocols allowing an arbitrary number of messages in transit.

The CFSM model uses explicit finite-state machines to represent processes and implicit queues to represent channels. The processes communicate by sending messages to one another via the channels. The queues modelling the channels have unbounded capacity to represent protocols allowing an arbitrary number of messages in transit.

More formally, a CFSM $M = (Q, q_0, \rightarrow)$ is a finite transition system where the following holds:

- Q is a finite set of *states* with $q_0 \in Q$ the *initial* state, and
- $\rightarrow \subseteq\ Q \times \mathcal{L} \times Q$; we write $q \xrightarrow{l} q'$ for $(q, l, q') \in \rightarrow$.

A CFSM $M = (Q, q_0, \rightarrow)$ is A-*local* if $\mathsf{sbj}(l) = \mathsf{A}$ for each $q \xrightarrow{l} q'$.

Given an A-local CFSM $M_\mathsf{A} = (Q_\mathsf{A}, q_{0\mathsf{A}}, \rightarrow_\mathsf{A})$ for each $\mathsf{A} \in \mathcal{P}$, the tuple $S = (M_\mathsf{A})_{\mathsf{A}\in\mathcal{P}}$ is a *(communicating) system*. For all $\mathsf{A} \neq \mathsf{B} \in \mathcal{P}$, it is assumed that there is a FIFO[3] queue b_{AB} of unbounded capacity, where M_A puts the message to M_B and from which M_B consumes the messages from M_A.

The semantics of communicating systems is defined in terms of *transition systems*, which keep track of the state of each machine and the content of each queue. Let $S = (M_\mathsf{A})_{\mathsf{A}\in\mathcal{P}}$ be a communicating system. A *configuration* of S is a pair $s = \langle \vec{q}\ ;\ \vec{b} \rangle$ where $\vec{q} = (q_\mathsf{A})_{\mathsf{A}\in\mathcal{P}}$ with $q_\mathsf{A} \in Q_\mathsf{A}$ and $\vec{b} = (b_{\mathsf{AB}})_{\mathsf{AB}\in\mathcal{C}}$ with $b_{\mathsf{AB}} \in \mathcal{M}^*$; state q_A keeps track of the state of the machine M_A and buffer b_{AB} keeps track of the messages sent from A to B. The

[2] Calculus of Communicating Systems
[3] First-In, First-Out

initial configuration s_0 is the one where, for all $\mathsf{A} \in \mathcal{P}$, q_{A} is the initial state of the corresponding CFSM and all buffers are empty.

A configuration $s' = \langle \vec{q}\,' \,;\, \vec{b}' \rangle$ is *reachable* from another configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ by *firing a transition* l, written $s \xrightarrow{l} s'$ if there is a message $\mathtt{m} \in \mathcal{M}$ such that either (1) or (2) below holds:

1. $l = \mathsf{A}\,\mathsf{B}!\mathtt{m}$ and $q_{\mathsf{A}} \xrightarrow{l}_{\mathsf{A}} q'_{\mathsf{A}}$ and
   a. $q'_{\mathsf{C}} = q_{\mathsf{C}}$ for all $\mathsf{C} \neq \mathsf{A}$ and
   b. $b'_{\mathsf{AB}} = b_{\mathsf{AB}}.\mathtt{m}$ and
   c. $b'_{\mathsf{CD}} = b_{\mathsf{CD}}$ for all $(\mathsf{C},\mathsf{D}) \neq (\mathsf{A},\mathsf{B}) \in \mathcal{C}$
2. $l = \mathsf{A}\,\mathsf{B}?\mathtt{m}$ and $q_{\mathsf{B}} \xrightarrow{l}_{\mathsf{B}} q'_{\mathsf{B}}$ and
   a. $q'_{\mathsf{C}} = q_{\mathsf{C}}$ for all $\mathsf{C} \neq \mathsf{B}$ and
   b. $b_{\mathsf{AB}} = \mathtt{m}.b'_{\mathsf{AB}}$ and
   c. $b'_{\mathsf{CD}} = b_{\mathsf{CD}}$ for all $(\mathsf{C},\mathsf{D}) \neq (\mathsf{A},\mathsf{B}) \in \mathcal{C}$

Condition (1) puts $\mathtt{m}$ on channel AB, while (2) gets $\mathtt{m}$ from channel AB. In both cases, any machine or buffer not involved in the transition is left unchanged in the new configuration s'.

A configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ is *stable* if all buffers are empty: $\vec{b} = \vec{\varepsilon}$. A configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ is a *deadlock* if $s \not\to$ and

- there exists a participant $\mathsf{A} \in \mathcal{P}$ such that $q_{\mathsf{A}} \xrightarrow{\mathsf{A}\,\mathsf{B}?\mathtt{m}}_{\mathsf{A}} q'_{\mathsf{A}}$
- or $\vec{b} \neq \vec{\varepsilon}$

This definition is adapted from [50] and is meant to capture communications misbehaviour. Observe that, according to this definition, a configuration s where all machines are in a state with no outgoing transitions and all buffers are empty is not a deadlock configuration even though $s \not\to$.

The language of a communicating system S is the set $\mathcal{L}[S] \in \mathcal{L}^\star$ of sequences $l_0 \dots l_{n-1}$ such that $s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$. Note that $\mathcal{L}[S]$ is prefix-closed.

Given two CFSMs $M = (Q, q_0, \to)$ and $M' = (Q', q'_0, \to')$, we use the following notations:

- $M \otimes M' = (Q \times Q', (q_0, q'_0), \to'')$ is the product of M and M' where $((q_1, q'_1), l, (q_2, q'_2)) \in \to''$ if, and only if,

$$((q_1, l, q_2) \in \to \text{ and } q'_1 = q'_2) \qquad \text{or} \qquad ((q'_1, l, q'_2) \in \to' \text{ and } q_1 = q_2)$$

- $\left\{ {}^{q'}/_{q} \right\} M$ represents the machine obtained by substituting the state q with the state q', provided that q' is not in the states of M;
- $M \otimes \mathbf{n}$ represents the machine $(Q \times \mathbf{n}, (q_0, n), \to \otimes \mathbf{n})$;
- $M \sqcup M'$ represents the machine $(Q \cup Q', q_0, \to \cup \to')$;

2.1.4 Well-formedness

Depending on the model of distribution, not all global specifications are able to be faithfully and correctly realized. Notions of *well-formedness* have therefore been defined by several authors so to guarantee sufficient conditions for global specifications to be realized in distributed settings. For our purposes, well-formedness reduces to *well-branchedness*, i.e., a condition that guarantees that distributed choices are "meaningful" in a distributed setting. The approach we introduce here is orthogonal to the notion of well-branchedness adopted; in fact, in Chapter 3 we give abstract requirements for well-branchedness. Hence, we sketch here a simple condition for well-branchedness[4] which, although not fully general, yields an instance of the abstract notion in Chapter 3.

A g-choreography is *well-branched* if for each choice, there is a unique *active* participant and any non-active participant is *passive* in the choice, where

- a participant is active when its first actions in each branch of the choice are different output actions, while
- a participant is passive when it is not involved in the choice or its first actions in each branch of the choice are different input actions.

This can be formalized relying on events that are minimal with respect to the partial order of pomsets:

Definition 2.1 (Active & passive participants)**.** Given a choice $\mathsf{G} = \mathsf{G}_1 + \cdots + \mathsf{G}_h$, let $L_{i,\mathsf{X}}$ be the set of actions which are the minimal events with subject X of $[\![\mathsf{G}_i]\!]$.

- A participant X is *active* in G if for $i \in \{1, \ldots, h\}$: $L_{i,\mathsf{X}}$ are pairwise disjoint non-empty sets of output actions.
- A participant X is *passive* in G if for $i \in \{1, \ldots, h\}$: $L_{i,\mathsf{X}}$ are pairwise disjoint sets of input actions (or they are all empty).

Example 2.4. *The simple g-choreography in Figure 1.1 is well-branched. In fact, as discussed earlier,* B *is the unique active participant (since its first actions are different outputs) while both* A *and* C *are passive since their first actions are different inputs.*

We now give a few examples of ill-branched g-choreographies.

Example 2.5. *We have that*

[4] Other examples of suitable notions of well-branchedness exist (e.g., [25, 48, 51]).

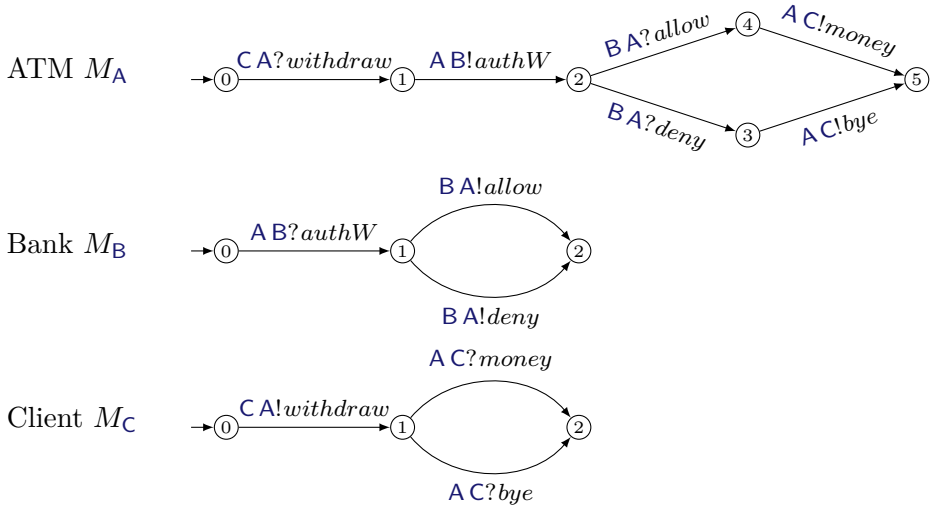


FIGURE 2.6: CFSMs for the protocol in Figure 1.1

- $\mathsf{G}_1 = \mathsf{A}\to\mathsf{B}\colon \mathtt{m} + \mathsf{A}\to\mathsf{C}\colon \mathtt{m}; \mathsf{A}\to\mathsf{B}\colon \mathtt{m}$ *is ill-branched*
- $\mathsf{G}_2 = \mathsf{A}\to\mathsf{C}\colon \mathtt{m} + \mathsf{B}\to\mathsf{C}\colon \mathtt{m}$ *is ill-branched.*

In fact, G_1 *has a unique active participant (*A*), but neither* B *nor* C *are passive (the former has the same input on both branches, while the latter has a non-empty set of actions only on the second branch);* G_2 *has two active participants (i.e.,* A *and* B*).*

Example 2.6. *A local view of the protocol from Figure 1.1 is given by the CFSMs in Figure 2.6. Notice how the events reflected in the global view have been split into their* send *and* receive *counterparts.*

The CFSM of the client initiates the protocol by sending a "withdraw" message to the ATM, which reacts by sending a message to the bank to check whether the client can actually perform this withdrawal. The CFSMs of the ATM and the bank will gradually proceed as they emit and consume messages from their queues.

State 1 *of the bank's machine is the* internal choice *state that corresponds to the branching point of the g-choreography. Namely, in state* 1 *the bank locally decides whether to allow or deny the client's request to withdraw money, respectively sending either an* `allow` *or a* `deny` *message. Accordingly, the ATM either delivers the* `money` *or finishes the conversation with a* `bye`*.*

Our framework is based on the analysis of traces possibly generated when putting the component under test in parallel with tests. This can be formalized in terms of runs of communicating systems. Also, our analysis considers "completed" runs, that is, those executions that reach a configuration where no further communications are possible. A standard way of formalizing these concepts is as follows.

Let $\mathcal{R}(S, s)$ be the set of *runs* of a communicating system S starting from a configuration s of S, that is, the set of sequences $\pi = \{(\hat{s}_i, l_i, \hat{s}_{i+1})\}_{0 \leq i \leq n}$ with $n \in \mathbb{N} \cup \{\infty\}$ such that $\hat{s}_0 = s$, and $\hat{s}_i \Longrightarrow_{l_i} \hat{s}_{i+1}$ for every $0 \leq i \leq n$; we say that run π is *maximal* if $n = \infty$ or $\hat{s}_n \not\Rightarrow$ and denote with $\mathcal{R}(S)$ the runs of S starting from its initial state.

The *language of a communicating system* S is the set

$$\mathcal{L}[S] = \bigcup_{\pi \in \mathcal{R}(S)} \{\text{trace of } \pi\}$$

where the *trace* of a run $\{(\hat{s}_i, l_i, \hat{s}_{i+1})\}_{0 \leq i \leq n} \in \mathcal{R}(S)$ is the sequence $l_0 \dots l_{n-1}$. Notice that systems have infinite and finite runs, in fact, $\mathcal{L}[S] \subseteq \mathcal{L}^{\omega}_{\text{act}} \cup \mathcal{L}^{\star}_{\text{act}}$ (where $\mathcal{L}^{\omega}_{\text{act}}$ is the set of infinite words over $\mathcal{L}_{\text{act}}$) and it is prefix-closed.

2.1.5 Projections

To link together both global and local views, in this work we rely on the notion of a *projection* operation. Although the conditions we give in Chapter 3 are abstract (and thus not tied to this specific operator), for the sake of clarity, in this section we introduce the operator we use in our implementations.

The basic idea of a projection operation is to associate a set of CFSMs representing the local behaviour of the participants of the choreography. We adapt a standard *projection* algorithm (see [48]) to generate communicating systems of CFSMs out of g-choreographies.

Before its formal definition, let us give an intuition of this construction, consisting basically of two steps. The first step inductively transforms each interaction of a g-choreography into the transition of an automaton according to a given role, say A, in that interaction. More precisely, the interaction becomes an output or an input transition depending on whether A is the sender or the receiver; otherwise the iteration corresponds to a silent transition. In the second step, the CFSM obtained as above is determinized. This is realized through an invariant maintained while projecting a g-choreography G on a participant A, that is, to single out two distinct states corresponding to the initial state q_0 of the machine, and a "gluing" state $q_e \neq q_0$ used to inductively combine projections.

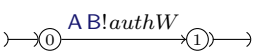
Pictorially, this can be represented as

where the arrows with the round tips mark the initial state and the gluing node.

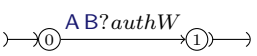
We focus on the cases when participant A occurs in the g-choreography G, since otherwise the projection consists of a CFSM without transitions and just a unique state (where q_0 and q_e "collapse").

The simplest case is the projection of an interaction involving A; the projection is a CFSM with a single transition labelled with the output part of the label if A is the sender or its input part if A is the receiver.

Example 2.7. *The projections of* A→B : `authW` *with respect to* A *and* B *and the states* 0 *and* 1 *are*



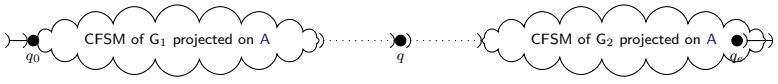
and



respectively.

The projection of the parallel composition is also straightforward: it is obtained by taking the product of the two CFSMs of G_1 and G_2 projected on A.

The projection of a sequential composition $\mathsf{G}_1; \mathsf{G}_2$ is simple and can be explained following this graphical representation:



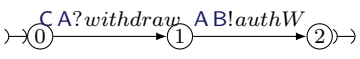
Essentially, one builds the CFSMs for G_1 and G_2 so that:

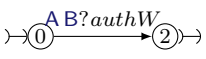
1. q_0 is the initial state of the CFSM for G_1, and
2. q_e is the gluing state of G_2.

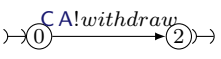
A fresh state q is used to connect the CFSM for G_1 and also used as the initial state of the CFSM for G_2 (as represented by the dotted lines connected to q).

In this way, whenever transitions of the first machine reach state q, transitions of the second machine can start, thus realizing the (expected) sequential behaviour.

Example 2.8. *Consider the g-choreography* C→A : `withdraw`; A→B : `authW` *consisting of the first two interactions of the example in Chapter 2.1.2. Then*

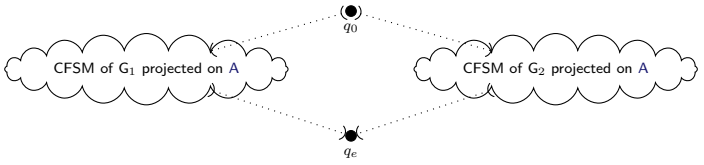






are the projections of A, B, *and* C *respectively.*

Projecting the branching of G_1 and G_2 is also simple, provided that the respective machines only share initial and gluing states. Consider the following graphical representation:

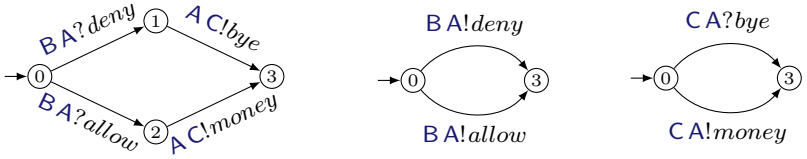


Namely, the resulting machine just follows either the executions of the machine of G_1 or the ones of the machine of G_2 just because the initial state is in common.

Example 2.9. *For the g-choreography*

$$\mathsf{B}\to\mathsf{A}\colon \texttt{deny};\mathsf{A}\to\mathsf{C}\colon \texttt{bye} \;+\; \mathsf{B}\to\mathsf{A}\colon \texttt{allow};\mathsf{A}\to\mathsf{C}\colon \texttt{money}$$

from the example of Chapter 2.1.2 we have that



are the projections of A, B, *and* C *respectively.*

The formal definition of the construction above is given as follows. Let q, q_0, q_e range over a fixed numerable set[5] that we use to build states of CFSMs as finite subsets of such set. Let G be a g-choreography, the function $\mathsf{G}\downarrow_{\mathsf{A}}^{q_0,q_e}$ yields the projection (in the form of a CFSM) of the choreography over the participant A using q_0 and q_e as initial and sink states respectively.

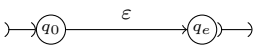
Before formally defining the projection function, we introduce some operations over CFSMs. Let $M = (Q, q_0, \to)$ and $M' = (Q', q_1, \to')$ be two CFSMs. The product of two CFSMs is defined as usual as $M \otimes M' = (Q \times Q', (q_0, q_0{}'), \to'')$ where $((q_1, q_2), l, (q_1', q_2')) \in \to''$ if, and only if,

$$((q_1, l, q_1') \in \to \text{ and } q_2 = q_2') \qquad \text{or} \qquad ((q_2, l, q_2') \in \to' \text{ and } q_1 = q_1')$$

We use $M \cap M'$ to denote the intersection of the states $Q \cap Q'$ and $M \sqcup M'$ to denote $(Q \cup Q', q_0, \to \cup \to')$.

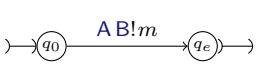
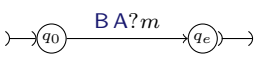
[5] Any infinite set can be chosen.

Definition 2.2 (Projection [48])**.** The *projection of* G *on* A *(and* $q_0 \neq q_e$*)*, symbolically $\mathsf{G}\downarrow_{\mathsf{A}}^{q_0,q_e}$, is obtained by *determinizing*[6] the CFSM defined as



if A does not occur in G

and the remaining cases are as follows:

$$\mathsf{G}\downarrow_{\mathsf{A}}^{q_0,q_e} = \begin{cases} \text{  } & \text{if } \mathsf{G} = \mathsf{A}{\to}\mathsf{B}\colon \mathtt{m} \\ \text{  } & \text{if } \mathsf{G} = \mathsf{B}{\to}\mathsf{A}\colon \mathtt{m} \\ \mathsf{G}_1\downarrow_{\mathsf{A}}^{q_0,q} \sqcup \mathsf{G}_2\downarrow_{\mathsf{A}}^{q,q_e} & \text{if } \mathsf{G} = \mathsf{G}_1;\mathsf{G}_2, \quad \text{and} \quad \mathsf{G}_1\downarrow_{\mathsf{A}}^{q_0,q} \cap \mathsf{G}_2\downarrow_{\mathsf{A}}^{q,q_e} = \{q\} \\ \mathsf{G}_1\downarrow_{\mathsf{A}}^{q_0,q_e} \sqcup \mathsf{G}_2\downarrow_{\mathsf{A}}^{q_0,q_e} & \text{if } \mathsf{G} = \mathsf{G}_1 + \mathsf{G}_2 \quad \text{and} \quad \mathsf{G}_1\downarrow_{\mathsf{A}}^{q_0,q_e} \cap \mathsf{G}_2\downarrow_{\mathsf{A}}^{q_0,q_e} = \{q_0, q_e\} \\ \mathsf{G}_1\downarrow_{\mathsf{A}}^{q_0,q_e} \otimes \mathsf{G}_2\downarrow_{\mathsf{A}}^{q_0,q_e} & \text{if } \mathsf{G} = \mathsf{G}_1 \mid \mathsf{G}_2 \end{cases}$$

where, given two CFSMs $M = (Q, q_0, \to)$, and $M' = (Q', q_1, \to')$, $M \sqcup M' = (Q \cup Q', q_0, \to \cup \to')$, with $q_0 \neq q \neq q_e$.

Note that in the case of sequential composition, the final state of G_1 is the initial state of G_2 and that the ε transitions are removed by determinizing.

2.2 Software Testing

The Guide to the Software Engineering Body of Knowledge [53] describes *software testing* as "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the expected behaviour."

In this definition, the specific meaning of *test case* depends greatly on the context we are working in. The typical, simpler case is for a test case to correspond to a pair of input-outputs. In the case of reactive systems, a test case can consist of a sequence of inputs and outputs, or even a tree or a graph for non-deterministic systems.

In other words, this usually means executing the program with specific input values (or sequences of values, i.e., traces) to find failures in its behaviour. One of the main advantages of this approach, when compared to static analysis, is that the program runs in an environment that is either the real environment or much closer to it. Hence, we don't only test the code, but also the compiler, libraries, networking and operating system support, for example.

[6] This can be done by applying classical algorithms like the partition refinement one [52] when interpreting CFSMs as finite automata on the alphabet $\mathcal{L}_{\text{act}}$.

Testing is not a silver bullet, however. In practical settings, testing cannot be exhaustive because of the large number of valid/invalid inputs that each operation can accept, as well as the fact that some systems can accept a potentially infinite sequence of operations. Also, traditional testing techniques rely on test cases which are usually handcrafted by developers. When applied correctly, testing has shown to increase the general quality and reliability of systems, but it still does not provide a guarantee of the correctness of the systems being tested.

There is also a challenge in choosing which tests to run from the (possibly) infinite set of test cases we could create. Ideally, we would want to prioritize those tests that are most likely to expose failures of the system. Although several techniques and strategies have been designed to deal with this issue, the problem is yet far from being solved.

Last but not least, after each test is executed, we must decide whether the observed behaviour was acceptable or not. This is called the *oracle problem*, and it is often solved via manual inspection of the test output.

There are several types of testing, which can be classified according to parameters such as the *scale* of the System Under Test, whether the tests are derived from requirements or code, and the specific characteristics of the system being tested [54]. According to scale, *unit testing* refers to the testing of a single unit (e.g., a procedure, a class, etc.). *Component testing* refers to testing each component or subsystem independently, whether *integration testing* intends to ensure that multiple components work correctly together. At a higher level, *system testing* checks the system as a whole to ensure it works correctly.

Testing can also be classified according to the specific characteristics being tested. The most common kind is *functional/behavioural testing*, where the search is focused on finding errors in the functionality of the system. *Robustness testing* is aimed towards finding errors in the system under invalid conditions, such as unexpected failures in the environment or incorrect inputs. *Performance testing* focuses on the performance of the system under heavy load, and finally, *usability testing* is focused on user interface problems that might affect the system's usability.

We consider our work closest to the categories of *conformance testing* [55] (a specific type of functional testing) and *specification-based testing* (as opposed to code-based testing) [56]. With code-based testing, either a formal or informal specification is used to derive a program, from which then tests are generated according to some adequacy criteria (such as coverage). In contrast, in specification-based testing both the program and the tests are derived directly from the specification. This means that tests can be created earlier

during development, and can be ready before the program is finished, enabling the test engineer to find inconsistencies and ambiguities in the specification and to correct them.

2.2.1 Model-based testing

A slightly more abstract approach to testing is *model-based testing*. The main assumption behind it is the existence of a precise formal model of the system being developed. This model can then be used in several ways to study the system in question. For instance, it can be used to generate suites of test cases to show conformance between the model and an actual implementation, or to generate a set of "interesting" test cases. This means that model-based testing can be used as a tool (or even complement) several of the testing types introduced in the previous section.

Model-based testing is mainly used to generate functional tests, but it can also be used for robustness testing, such as testing the system with invalid inputs. It can also be used for performance testing (although this is an area more currently under development [10]). It is also considered a form of black-box testing, since the tests are generated from a specification of the system, instead of being directly obtained from the code.

The model can also be used as part of a *model checking* approach: in the classic approach, specifications are expressed in a propositional temporal logic, and the reactive system is modelled as a state-transition graph. An efficient search procedure is used to determine automatically if the specifications are satisfied by the state-transition graph [57].

Utting and Legeard argue that the term *model-based testing* has encompassed the following approaches in the literature [10]:

1. generating test input data from a domain model,
2. generating test cases from an environment model,
3. generating test cases with oracles from a behaviour model,
4. generating test scripts from abstract tests.

In the first approach, the model provides information on the domains of the input values: generating test input data consists of selecting and combining automatically values drawn from these domains. This is useful, but it does not solve the test design problem completely since it does not help us know if a test has passed or not.

The second approach uses a different type of model, one that describes the expected environment of the System Under Test (SUT), such as a statistical model of the expected usage of the system. This has the disadvantage that the output values cannot be predicted because the behaviour of the system is not included in the model. For this reason, this approach is commonly used only to detect whether the system has failed (e.g., crashed).

The third approach refers to the generation of executable test cases that include *oracle information* (i.e., which output values are expected to be obtained from the system after execution). This is intrinsically more difficult because the model must contain enough information about the expected behaviour of the SUT to be able to predict and check the output values. The advantage of this approach is that this is the only one that covers the complete test design problem, from choosing input values to generating executable test cases with information about the outcomes.

The fourth approach focuses on transforming very abstract descriptions of test cases (such as a UML[7] sequence diagram) into a low-level executable test script. In this case, the model is information about the structure and API[8] of the SUT, as well as details on how to transform a high-level call into executable test scripts.

In software testing, an important distinction exists between software fault, error, and failure. A software fault refers to a static defect inherent within the software itself. An error, on the other hand, represents an incorrect internal state which is a manifestation of a software fault. Finally, a software failure is the observable, external behaviour that deviates from the expected output or requirements, arising from an internal error.

Analysing these ideas leads to the "RIP" model, which states that three conditions must be present for a failure to be observed [58]:

Reachability The location or locations in the program that contain the fault must be reached.

Infection After executing the location, the state of the program must be incorrect.

Propagation The infected state must propagate to cause some output of the program to be incorrect.

The RIP model holds significant importance in the domain of coverage criteria, particularly for mutation testing and automatic test data generation. Interestingly, despite its

[7] Unified Modelling Language
[8] Application Programming Interface

universal acceptance, the terminology within the RIP model was developed independently by different researchers. While Morell used the terms *execution*, *infection*, and *propagation*, Offutt referred to them as reachability, sufficiency, and necessity [59, 60].

2.2.2 Property-based testing

Traditional unit testing is commonly based on providing a set of inputs to a function, and then checking that the function returns the expected output. This approach is also sometimes referred to as "example-based testing", and often requires the developer to manually provide a set of inputs and the corresponding expected outputs, i.e., directly deal with the "oracle problem" of determining whether the output is correct.

Designing tests that use these input-output pairs is often a cumbersome process that requires a lot of manual effort: it tends to be difficult to come up with a set of inputs that are representative of the possible inputs that the function can receive, and, depending on the complexity of the code being tested, it might also be difficult to come up with the corresponding correct outputs.

Property-based testing is an alternative approach that is based on specifying properties that a program should satisfy, and then generating random inputs that are used to test a program. It is similar to fuzzing, in the sense that random inputs are generated, but fuzzing is usually focused on finding whether the code crashes, while property-based testing tends to be focused on testing whether the code satisfies a more general set of properties. This approach is beneficial because it can uncover bugs and edge cases that may not have been considered in manual testing. It also allows for more comprehensive testing and can reduce the number of test cases needed to achieve high coverage. Additionally, property based testing can help improve the design of the program by encouraging the developer to think in terms of general properties that the program should satisfy, rather than specific inputs and outputs [61].

Property-based testing has a strong relationship with model-based testing. Both methodologies share a common emphasis on higher-level abstractions about the system's behaviour, rather than focusing on specific, concrete test cases. In model-based testing, a model of the system's behaviour is used to generate test cases. Similarly, in property-based testing, properties that are abstract descriptions of the system's behaviour are used to generate test cases. This testing methodology involves defining general properties that the system under test should uphold, rather than specifying individual test cases with expected outcomes. Instead of manually crafting each test case, a property-based testing framework generates many random inputs and checks if the system maintains the defined properties for all these inputs. If any violation of the properties is found, the

framework often provides a minimal failing case, which can be helpful in diagnosing the issue. This approach contrasts with traditional example-based testing, where specific inputs and expected outputs are manually defined by the tester. Property-based testing allows for a more comprehensive exploration of the input space, increasing the chances of uncovering edge cases or rare bugs that may not have been considered in manual test case design.

Hypothesis is, at the time of writing, the most popular property-based testing library for Python [62, 63]. It provides a variety of built-in data types and *strategies* for generating inputs. A strategy can be defined as a formalized description for generating test cases. In this sense, a strategy essentially outlines the data space that the testing system should explore. The description does not only encompass the type or class of the data, but also other associated attributes such as bounds, length, or distribution of values, among others. This makes strategies a potent tool for producing inputs that are representative of the varied data that the software could encounter in real-world usage. Hence, they enable the discovery of edge cases that may not be typically accounted for in manual testing.

Hypothesis offers numerous built-in strategies for common data types and structures. For example, `integers()`, `lists()`, `floats()`, `text()`, and `dictionaries()` are among the built-in strategies. Each of these comes with parameters that can be used to further customize the generated data. Furthermore, the library also allows for the creation of custom strategies, thus providing flexibility to cater to more complex testing scenarios. It also supports defining custom strategies for generating more complex data types [63].

A key feature of Hypothesis is its ability to *shrink* inputs to find minimal examples that still violate the properties. It uses search strategies to find the minimal inputs that uncover a given bug [61]. This shrinking process makes it easier to understand and reproduce bugs, which can radically simplify the debugging process.

Hypothesis also offers support for testing stateful systems with a state machine abstraction. This allows the tester to define a set of actions that can be performed on a system and generate random sequences of actions for testing. If a sequence of actions leads to a state that violates a property, Hypothesis will try to find a minimal sequence of actions that results in the same state and report this sequence as the failing test case [64]. This test case is saved and prioritized in subsequent test runs to verify that the bug has been fixed.

2.3 Related Work

There is a lot of academic research on using behavioural types to verify properties of message-passing systems, as well as model-based testing in general, albeit these two fields of research are not often combined. In this section we give just a few examples, mentioning tools that span across different mainstream languages, as well as the properties they intend to verify or enforce. We also mention a few model-based testing approaches that have similar approaches to ours.

CFSMs have been a pioneer theoretical formalism, allowing for the study of distributed safety properties [46, 49]. Scalas and Yoshida [65] argue that the safety of their interactions (that is generally undecidable) is verified with two main approaches: (a) assuming the decidability of a synchronisability[9] property [66], and then checking temporal properties of CFSMs via model checking; or (b) checking decidable synchronous execution conditions on CFSMs, and prove that they ensure safe asynchronous executions [67]. Synchronisability is non-decidable in general according to Finkel and Lozes [66, 68], so the first variant is not generally feasible. This has been just one of the motivations for further developments in the MPST theory. For instance, Scalas and Yoshida have proposed a generalized MPST theory [65], claiming it to be less complex and more general than the classical framework [69]. Their most recent models sidestep several requirements (e.g., projections, duality/consistency) of more traditional MPST frameworks and enhance type safety for further protocols and processes. Through the use of parameterized types, they are able to ensure that type-checking is decidable, and that processes are safe, deadlock-free and live. In our work, however, we concern ourselves with unbounded systems, which can easily simulate Turing Machines, and thus do not require type-checking to be decidable.

Tuosto and Guanciale have introduced formal semantics for global graphs, presented as partially ordered multisets of communication events (pomsets) and based on hypergraphs of events [48]. They argue that, while the former semantics is more elegant, the latter is more suitable for implementation[10]. The ChorGram tool, developed by these authors, consists of a tool chain to support choreographic development of message-oriented applications [70]. The tool chain also has an implementation of the algorithm to derive a global graph from CFSMs, and can verify property and reconstruct a global graph from a given valid specification satisfying Generalized Multiparty Compatibility [70].

[9] The intuitive notion behind synchronisability is that an asynchronously communicating system is synchronisable if executing that system with synchronous communication instead of asynchronous communication preserves its behaviours.

[10] For the purposes of this work, we focus on the former, as it is more intuitive and easier to understand.

Among the most common notations to represent choreography models, one of the most well-known ones is BPMN (Business Process Model and Notation), a standardized, graphical notation for drawing business processes in a workflow [12, 71–73]. It was developed by the Business Process Management Initiative (BPMI) and is currently maintained by the Object Management Group (OMG). BPMN's main objective is to provide a notation that is readily understandable by all business stakeholders. These business stakeholders include the business analysts who create and refine the processes, the technical developers responsible for implementing the technology that will perform those processes, and the business managers who monitor and manage the processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation. It is worth noting that the complexity of BPMN is a double-edged sword: while it is very expressive, it is also plagued with ambiguities and as such, there have been different attempts to give it a formal semantics [73–75].

BPMN provides three types of diagrams to represent business processes: Process Diagrams, Collaboration Diagrams, and Choreography Diagrams:

Process Diagrams depict a sequence of activities that represents how work is done within an organization or a system. This sequence of activities is typically presented as a flowchart, and it illustrates the flow of control from one activity to the next within a particular participant or role.

Collaboration diagrams are an extension of the process diagram. They show interactions between different participants in a process, with each participant represented as a swimlane (or a "Pool"). These are akin to what we consider "local views" in our work.

Choreography diagrams focus on the exchange of information (messages) between participants rather than the actual work done or the sequence of activities. They represent a sequence of activities that details how participants coordinate their interactions [76].

The ParTes approach [77] uses an *interaction tree* to generate tests out of BPMN2 diagrams. From this tree, they use a projection operation to generate a *trace snippet*, which they combine to form a *participant interaction tree*. This tree is then reorganized and used to generate test skeletons and dynamically reconstruct execution traces. They define a sequence of reductions to try to mitigate the state explosion problem. Their approach is somewhat similar to ours, but (being based on BPMN) lacks a formal semantics, and we believe our notion of projection to be more general. ParTes has been used as part of a larger framework for online testing of choreographed services, in the

context of the CHOReOS European Project [78]. Among others, this work sketches a test generation procedure which is however not supported by a formal semantics as we do.

Along this line, another interesting work is the one of Zhou et al. [79] which propose a method based on *dynamic symbolic execution*[11] to test web services by adding assertions on data to standard WS-CDL choreography specifications. Their intuition is to execute the choreography in a simulation engine based on symbolic execution and use an SMT-solver[12] (specifically Z3) to find values that satisfy the path conditions of each execution. By iteratively negating the last condition in the execution path, new test input data are generated in order to cover the whole choreography.

In [80, 81] the testing theory of [82] has been used as the ground for *ioco-testing* ("input-output compliance") of communicating processes. This setting relies on labelled transitions systems (formalized through an operational semantics) to verify whether processes conform to some input-output behaviour. The ioco-testing theory advocates for finite and deterministic tests [81] in the context of synchronous communications, which we also do for asynchronous communications. A key difference is that the use of g-choreographies allows us to avoid using the rather "invasive" observing mechanism of the ioco-theory [80].

Tools based on Multiparty Session Types have also found their way into mainstream programming languages. For example, Lange et al. [83] have proposed a method to statically verify liveness and communication safety under certain conditions, using the Go programming language as a case study. They formalize a subset of the Go language as a process calculus, introduce a typing system for it. They subsequently define a framework based on the notions of *symbolic execution* and *fenced types*, the latter a concept they coin in the same paper. Following this work, Lange et al. [84] have also introduced a static analysis tool called Godel Checker, which can automatically verify some safety and liveness errors from Go programs. Their process works by inferring behavioural types from source code and using a model checking tool (mCRL2 [85]) in order to check for properties such as the absence of global deadlocks.

The SPY (Session Python) toolset proposed by Neykova et al. [39] uses runtime monitoring to verify distributed Python programs against Scribble protocol specifications. The tool is based on a specification language with a formal semantics and requires no synchronization between monitors. It has also been applied in industry as part of the Scribble-OOI collaboration [86].

[11] The idea behind *symbolic execution* is executing a given program with symbolic values instead of actual values, and can support several tasks such as automatic test case generation and coverage analysis.

[12] Satisfiability Modulo Theory

The Haskell language has also been target to several implementations of session types. Orchard and Yoshida [87] have proposed embedding session types into an effect system[13] in Haskell.

Laumann et al. have proposed as well an implementation of session types for the Rust language [41]. They make use of Rust's affine types[14] to guarantee a linear usage of communication channels (i.e., that they're used exactly once, avoiding aliasing) and argue that affine types are sufficient for preventing protocol violations. The library resulting from this research was used in the Servo experimental browser engine to describe internal communication patterns [41]. This particular implementation strongly inspired the one we introduce in Chapter 4.

Other models that have been used for testing include Message Sequence Charts (MSC), which depict the exchange of messages between the processes of a distributed system over a single partially ordered execution [88]. Sequence diagrams are a particular type of MSCs, and they have been heavily adopted by the industry as part of the UML standard. Since they give a broad overview of how a whole system is supposed to work (generally from an abstract point of view) we can consider them as a global model. In many object-oriented system development methodologies, the user first specifies the system's use cases and some specific instantiations of each use case are then described using sequence diagrams. In a later modelling step, the behaviour of a class is described by a state diagram (usually a statechart) that prescribes a behaviour for each of the instances of the class. Finally, the objects are implemented as code in a specific programming language. Parts of this design flow can be automated, and the main role of MSCs is to capture system requirements in the form of scenarios that the implemented system should exhibit.

Micskei and Waeselynck [89] remark that sequence diagrams changed significantly with UML 2.0, arguing that their expressiveness (from an engineering point of view) was highly increased. This also increased language complexity, yielding several possible choices in its semantics. They surveyed available formal semantics for sequence diagrams and explained how these approaches handle the different semantic choices. Some of those semantics might be more suitable as foundational theories for model-based testing.

Harel and Marelly [90] argue that sequence charts possess an extremely weak partial-order semantics that does not make it possible to capture interesting behavioural requirements of a system. They state that sequence charts are far weaker than, e.g., temporal logic or other formal languages for requirements and constraints. To address this, an

[13] Effect types are another kind of behavioural types, based on λ-calculus, instead of the π-calculus on which session types are based.

[14] An *affine type* in Rust can be used once or not at all. This restriction helps maintain memory safety.

extension of the language of MSCs was proposed in 1999, called live sequence charts (LSCs) [91]. LSCs distinguish between scenarios that *may* happen in the system (existential charts) from those that *must* happen (universal charts). Also, they can specify messages that *may* be received (cold) and ones that *must* (hot). This increased expressiveness makes it possible to better grasp the relationships between possible behavioural artefacts of these models.

To mention a few tools with similar approaches to testing reactive systems, Modbat is a model-based API testing tool that utilizes extended finite-state machines to model system behaviour using a domain-specific language embedded on top of Scala [92]. An Extended Finite State Machine (EFSM) is a generalization of the traditional finite state machine model. In essence, the EFSM model can be viewed as a compact representation of a machine where data is not a part of the state space; data is handled separately, and operations on data are modelled in the state transitions. A block in an EFSM represents a set of states, retaining thus many advantages of the finite state machine model while attempting to overcome the major limitation of traditional finite automata — state explosion [93]. Modbat simplifies the test modelling process, with a tester defining a model and running Modbat against it. It explores the model's possibilities using a random search and executes the system under test (SUT) in tandem, resetting both after each test. Modbat has proven effective in uncovering previously unknown defects in complex systems due to its ability to handle non-deterministic operations and exception handling [94].

Another interesting tool is P# (a.k.a. Coyote), an open-source, actor-based .NET programming framework [95]. Each actor operates concurrently with others, communicating via event messages and managing its own state, thereby eliminating shared state and limiting synchronization to event sending. This approach improves upon traditional threads and locks models by clearly marking communication points between actors in code. P# requires developers to explicitly declare all non-determinism in their code, enhancing the robustness of non-deterministic system testing. The P# tester understands the non-determinism arising from concurrency between actors and employs hooks into the P# runtime to control actor scheduling. Additionally, P# provides an API for generating unconstrained Boolean and integer values to model non-determinism and supports state-of-the-art search strategies. It also facilitates the writing of functional specifications via monitors, observing program execution without influencing it, and enabling assertions that span multiple actors, including liveness properties to ensure system progress. Consequently, P# development involves system creation using its concurrency model, mocking external dependencies and expressing all non-determinism, and writing tests and specifications for asserting correctness. P# has been used to test components of the Microsoft Azure cloud platform, uncovering several previously unknown bugs [96].

Although in our work we focus on black-box testing, it is worth mentioning the work on SAGE [97], a white-box testing tool for finding concurrency errors in multithreaded programs. SAGE, introduced by Microsoft in 2007, performs dynamic symbolic execution at the binary (x86) level, making it agnostic to the programming language or build process used. Its capabilities allow it to detect security vulnerabilities that other methods, such as static program analysis and black-box fuzzing, often miss. By identifying these vulnerabilities before software release, SAGE has helped avoid the need for costly security patches. Its capacity to handle large-scale execution traces and its array of optimizations for speed and memory efficiency have made it a valuable asset in verifying large applications. Due to its successful bug detection and ease of use, SAGE operates continuously across various groups within Microsoft [98].

Chapter 3

An Abstract Framework for Choreographic Testing

We now give the formal constructions of our approach. After introducing our basic ingredients in Chapter 3.1, we give our test generation algorithm in Chapter 3.2. Chapter 3.3 discusses the implementation of the algorithm and the command-line tool we developed to support it, and introduce a larger version of the ATM in Chapter 3.4, explaining how the proposed algorithm works applied to this case. We also validate the approach by means of an implementation at the abstract level, and discuss its effectiveness through an analysis based on mutation testing in Chapter 3.5.

3.1 Introduction

As remarked in [49], "safe" computations of communicating systems necessarily reach configurations that are *stable*, namely configurations whose buffers are all empty (note that stability does not impose any requirement on a machine's enabled transitions). We adapt this notion to our testing framework by allowing the non-emptiness condition to apply only to a subset of the channels: a configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ is *stable* for $\mathcal{C}' \subseteq \mathcal{C}$ if all buffers in $\mathcal{C}'$ are empty in s. Moreover, we consider a test that "cannot be completed" as failed.

Definition 3.1 (Unsafe configuration)**.** A configuration $s = \langle \vec{q} \,;\, \vec{b} \rangle$ is *unsafe* if $s \not\Longrightarrow$ and either there is a participant $\mathsf{A} \in \mathcal{P}$ such that $\vec{q}(\mathsf{A}) \xrightarrow{\mathsf{A}\,\mathsf{B}?\mathtt{m}}_{\mathsf{A}}$ or s is not stable.

This definition is adapted from [50] and is meant to capture communication misbehaviour. Observe that, according to this definition, a configuration s where all machines

are in a state with no outgoing transitions and all buffers are empty is not unsafe even though $s \not\Longrightarrow$.

Our approach to generate tests reuses ingredients of existing top-down approaches of choreographies. These usually define projection functions (like the one presented in Section 2.1.5) that generate local models from global ones. In order to parameterize our framework with respect to these notions, we introduce the notion of *abstract projections* on g-choreographies.

Definition 3.2 (Abstract projection)**.** A map $\_\restriction\_$ is an *abstract projection* if it takes a g-choreography G and a participant $\mathsf{A} \in \mathcal{P}$ and returns an A-local CFSM. Given a g-choreography G, the *system induced by* $\_\restriction\_$ is defined as $\mathsf{G}\restriction = (\mathsf{G}\restriction_{\mathsf{A}})_{\mathsf{A}\in\mathcal{P}}$.

Definition 3.2 encompasses several "concrete" projection operations. For example, the operation in Section 2.1.5 is an instance of an abstract projection, which we adopt in our examples.

Not every g-choreography can be *faithfully* projected. In fact, the distributed execution of communicating machines can exhibit different behaviour than the one specified in the g-choreography. In concrete instances, sufficient conditions are given so that the semantics of projected communicating systems reflects the semantics of the corresponding g-choreography. These conditions are abstractly captured in the next definition.

Definition 3.3 (Abstract well-formedness)**.** A communicating system S *realizes* a g-choreography G when

- $\mathcal{L}[S] \subseteq \mathcal{L}[\mathsf{G}]$ and
- no run in $\mathcal{R}(S)$ contains an unsafe configuration.

A predicate on g-choreographies WF is an *abstract well-formedness condition* if $\mathit{WF}(\mathsf{G})$ implies that there is a communicating system S that realizes G.

It is worth noting that Definition 3.3 regards communicating systems S realizing a g-choreography G as *refinements* of G. Therefore, Definition 3.3 requires only that the executions of S are included in the ones specified by (the semantics) of G (in other words, implementations show "less behaviour" than their specification). Note that this implies the correctness of S (an implementation does not introduce undesired behaviour) with respect to G but not its completeness. This is useful when the communication mechanism adopted at lower levels of abstraction has restrictions not present in the semantics of the choreography (e.g., limited size buffers, ordered buffers, etc.).

The short ATM g-choreography from Chapter 2.1.2 is considered well-formed in the majority of existing work (e.g., [25, 99, 100]). In it, there is only one participant that makes a choice (i.e., the bank) and the rest of the participants are informed of which decision was taken. Intuitively, this avoids coordination problems, and therefore the choreography can be correctly realized by CFSMs, such as the ones in Figure 2.6. In this case, the language of the choreography is the same as the language of the projected communicating system; notice that the system does not reach unsafe configurations.

Hereafter, we assume projections that *respect* abstract well-formedness.

Definition 3.4 (Compatible projections)**.** An abstract projection $\_\upharpoonright\_$ is *compatible with* WF when, for all g-choreographies G, if $\mathit{WF}(\mathsf{G})$ then the system induced by $\_\upharpoonright\_$ realizes G.

An abstract projection mapping all participants to a machine without any transitions is trivially compatible with any abstract well-formedness condition. Of course, we are interested in abstract projections for which $\mathcal{L}[\mathsf{G}\upharpoonright] = \emptyset$ only if $\mathsf{G} = (\mathsf{o})$. We remark that the projection operation in Chapter 2.1.5 is compatible with the notion of well-formedness discussed previously (cf. [101]).

We can now formalize the main notions of our framework: what is a test case, how tests are executed, when an execution passes a test, and when a test is meaningful for a choreography.

A *test case* for a Component Under Test (CUT) A is a set of deterministic CFSMs that can interact with A, together with a distinct set of "success" states.

Definition 3.5 (Test case)**.** Let $\mathcal{L}^{!}_{\mathrm{act}} \subseteq \mathcal{L}_{\mathrm{act}}$ be the set of output actions.

A set $T = \bigcup_{1 \leq i \leq n} \{\langle M_i, \underline{Q}_i \rangle\}$ is a *test case for a CUT* $\mathsf{A} \in \mathcal{P}$ if for every $1 \leq i \leq n$, $M_i = (Q_i, q_{0i}, \rightarrow_i)$ is a CFSM with $\underline{Q}_i \subseteq Q_i$ and

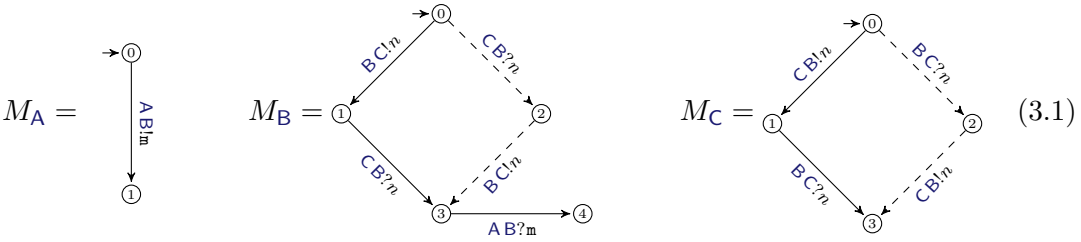
1. if $q \xrightarrow{l}_i q'$ then $\mathsf{sbj}(l) \neq \mathsf{A}$
2. if $q \xrightarrow{l}_i q'$ and $q \xrightarrow{l'}_i q''$ then $(l \in \mathcal{L}^{!}_{\mathrm{act}} \implies l = l')$ and $(l = l' \implies q' = q'')$
3. if $q_1 \xrightarrow{l}_i q_2$ and $q'_1 \xrightarrow{l'}_j q'_2$ with $j \neq i$ then $\mathsf{sbj}(l) \neq \mathsf{sbj}(l')$

We dub $\underline{Q}_i$ the *success states* of M_i.

We briefly justify the conditions in Definition 3.5. Condition (1) forces the CUT not to be the subject of any transition, since tests cannot force it directly to take specific actions. Condition (2) enforces that there is always a single possible output for the

system to proceed, that the machines are deterministic and, in particular, that they cannot have internal choice or mixed-choice[1] states. The rationale behind condition (2) is to "confine" non-determinism to the CUT and its concurrent execution with the test so that it is easier to analyse the outcome of tests. Condition (3) enforces transitions across machines to have different subjects: if this were not the case, generating code for each participant could be significantly more complex. Note that this does not force the CFSMs in a test case to be necessarily local; in fact, Definition 3.5 admits different subjects in the labels of different transitions.

Ruling out *mixed-choice* states from test CFSMs is a design choice because otherwise they would introduce non-determinism in the test and hinder repeatability. We use a conservative approach and in a mixed-choice state, say q, we select a test starting with one of the output transitions of q and drop all the others. This is illustrated with a simple communicating system consisting of the following CFSMs:



(3.1)

where M_{A} is the CUT. In this case, we generate one single test using the machines resulting from dropping the dashed edges. Notice that the alternative approach, i.e., including the dashed transitions instead of the solid ones, has a run to a deadlock configuration despite the fact that M_{A} behaves as expected. That is, since our framework is asynchronous, and we consider channels of unbounded capacity, all the machines can initially send even when no other is receiving; if both machines decided to receive first instead, this would result in a deadlock.

The previous argument is interesting, since it allows us to reduce the number of transitions in the resulting test machines without affecting coverage at the choreography level. In other words, it is possible to deal with mixed choices with further insight on the actual notion of well-formedness and of the projection operation. In particular, the well-formedness condition and the projection operation that we adopt ensure that mixed-choice states occur in projected machines only for parallel g-choreographies. This allows us to show that the discarded tests (i.e., the ones that use inputs of mixed-choice states) are indeed redundant. Intuitively, this redundancy is due to the fact that changing the order of these send and receive actions in a test machine is irrelevant to the CUT. In fact, note that in this case projections would yield machines where the output

[1] A mixed-choice state is one with both input and output outgoing transitions.

and input actions can happen in any order. Therefore, the "branch" where the input takes place after the output subsumes the other branch.

The following example shows the requirements of Definition 3.5 and a violation of those requirements.

Example 3.1. *Consider the CFSMs M_{A}, M_{B} and M_{C} in Figure 2.6. A test case for* B *(i.e., the bank) is $T_1 = \{\langle M_{\mathsf{A}}, \{5\}\rangle, \langle M_{\mathsf{C}}, \{2\}\rangle\}$. In fact, M_{A} and M_{C} are deterministic (hence internal choice-free) and do not include any transitions where the subject is* B*. Instead, $T_2 = \{\langle M_{\mathsf{B}}, \{2\}\rangle, \langle M_{\mathsf{C}}, \{2\}\rangle\}$ is not a test case for* A *(i.e., the ATM) because M_{B} has an internal choice in state* 1*.*

A test is executed on a CFSM $\hat{M}$ by simply running it in parallel with $\hat{M}$. The outcome of the test is determined by inspecting maximal runs of such execution. This is formalized in the next definition.

Definition 3.6 (Test compliance)**.** Let $\mathcal{C}' \subseteq \mathcal{C}$ be a set of channels, $\hat{M}$ an A-local CFSM, and T a test case for A. Denote with $\hat{M} \otimes T$ the communicating system consisting of $\hat{M}$ and the CFSMs in T.

Machine $\hat{M}$ is *T-compliant with respect to $\mathcal{C}'$*, symbolically $\hat{M} \triangleright_{\mathcal{C}'} T$, if every maximal run of $\hat{M} \otimes T$ contains a stable configuration s for $\mathcal{C}'$ such that for every $\langle M, \underline{Q}\rangle \in T$ the local state of M in s is in $\underline{Q}$.

Intuitively, test compliance requires that to pass a test, a CUT should not lead the system into configurations where messages are not "properly consumed" or some of CFSMs of the test are not in an "expected state". In the following, we dub the configuration s in Definition 3.6 a *successful configuration for T*, and we use $\hat{M} \triangleright T$ for $\hat{M} \triangleright_{\mathcal{C}} T$. Notice that the parametrization on $\mathcal{C}'$ allows a CFSM to be considered compliant even if some runs leave channels in $\mathcal{C} \setminus \mathcal{C}'$ not empty. The next series of examples illustrate the notion of test compliance with four tests for CUTs in Figure 2.6.

Example 3.2. *With reference to Example 3.1, consider the test case T_1 and assume M_{B} to be the CUT. The CFSM M_{B} in Figure 2.6 is T_1-compliant. In fact, the system consisting of M_{B} and (the CFSMs in) T_1 is exactly the system implementing the choreography of the running example.*

Example 3.3. *Let $T_3 = \{\langle M_{\mathsf{A}}, \{2\}\rangle, \langle M_{\mathsf{C}}, \{2\}\rangle\}$, M_{B} is not compliant with T_3. This is due to the fact that* C *can reach* 2 *only after that* A *has left state* 2*. Also, let $T_4 = \{\langle M'_{\mathsf{A}}, \{5\}\rangle, \langle M_{\mathsf{C}}, \{2\}\rangle\}$, where M'_{A} is obtained by removing the transition* B A!*allow from M_{A}. M_{B} is not compliant with T_4 since there is a run without stable configuration: i.e., when* B *decides to send* `allow`*.*

Example 3.4. *Suppose that the CUT is the CFSM M'_{B} obtained by removing the transition* $\mathsf{B}\,\mathsf{A}!$*allow from* M_{B}*. Then* $M'_{\mathsf{B}} \triangleright T_1$*.*

Example 3.5. *Let M_{A} be the CUT and M'_{B} be the CFSM obtained by removing the transition* $\mathsf{B}\,\mathsf{A}!$*allow from* M_{B}*. Then M_{A} is compliant with* $\{\langle M'_{\mathsf{B}}, \{2\}\rangle, \langle M_{\mathsf{C}}, \{2\}\rangle\}$*.*

We finally define when a test case is meaningful for a choreography, by requiring that the correct realization (i.e., the projection) of the choreography is compliant with the test.

Definition 3.7 (Test suitability)**.** Test T is $(\mathsf{G}, \mathsf{A}) - \textit{suitable}$ if $\mathsf{G}\!\downarrow_{\mathsf{A}} \triangleright T$.

Notice that all tests in Examples 3.2 and 3.5 are suitable for the choreography used in the introduction, since the projection is compliant with them. On the other hand, tests of Example 3.3 are not suitable for the same choreography.

3.2 Test Generation Algorithm

To generate tests we follow a straightforward strategy: we start from the projections of the participants (other than the CUT) and we split their internal choices. The intuition is that a projection of a well-formed g-choreography is "compatible" with any implementation where only the CUT performs internal choices.

We use the following auxiliary function to identify non-deterministic states. These are the states that the algorithm uses to split the transitions to obtain deterministic tests. Given a CFSM $M = (Q, q_0, \to)$, let

$$\mathit{nds}(M) = \left\{ q \in Q \mid \exists q \xrightarrow{l_1} q_1 \neq q \xrightarrow{l_2} q_2 \,:\, l_1 = l_2 \vee \{l_1, l_2\} \cap \mathcal{L}^!_{\mathrm{act}} \neq \emptyset) \right\}$$

be the set of non-deterministic states of M, that is the states with at least two different transitions that either have the same label or one of which is an output transition. For convenience, we let $M(q)$ denote the set of outgoing transitions of q in M and $M - t$ (resp. $M + t$) be the operation that removes from (resp. adds to) M transition t (these operations extend element-wise to sets of transitions). The following function produces

sets of machines that are internal choice free:

$$\mathsf{split}(M) = \begin{cases} \{M\} & \text{if } \mathit{nds}(M) = \emptyset \\ \bigcup_{q \in \mathit{nds}(M)} \mathsf{split}(M, q) & \text{otherwise} \end{cases}$$

$$\mathsf{split}(M, q) = \begin{cases} \bigcup\limits_{q \xrightarrow{\mathsf{A\,B!m}} q'} \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} q') & \text{if } M(q) \text{ has output transitions} \\ \bigcup\limits_{\substack{q \xrightarrow{\mathsf{A\,B?m}} q' \\ \neq \\ q \xrightarrow{\mathsf{A\,B?m}} q''}} \mathsf{split}(M - q \xrightarrow{\mathsf{A\,B?m}} q') & \text{otherwise.} \end{cases}$$

Notice that, for the sake of clarity, this definition takes the union over all non-deterministic states of M. This avoids the need of defining an order of non-deterministic states and simplifies verification of Theorem 3.11, but can lead to the introduction of many redundant tests. In fact, the split operation is insensitive to which non-deterministic state of M is chosen at each step. This property is formalized in Chapter 3.3 by Theorem 3.12 and is used to optimize the implementation.

Once these simpler CFSMs are obtained, success states have to be set for each of them. This is analogous to the problem commonly known in software testing as the *oracle problem*: deciding when a test is successful. This decision is application-dependent and its solutions usually require human intervention [102]. In our setting, this corresponds to pinpointing configurations of communicating systems, according to a subtree of a choreography as we define later on. Intuitively, we would like success states from the CFSMs to correspond to the execution of specific syntactic subtrees of the choreography.

We now introduce an additional definition (3.9) that helps us determine the success states for our tests. In the following, given a g-choreography G, let $\mathbb{T}(\mathsf{G})$ be the set of subtrees of the abstract syntax tree producing G once we fix a suitable precedence among the operators[2].

For the following definitions, consider:

- G a g-choreography,
- $\_\upharpoonright\_$ an abstract projection compatible with a given well-formedness predicate WF,
- S the communicating system induced by $\_\upharpoonright\_$,
- $\Omega_{\mathsf{G},\upharpoonright}$ a map assigning a set of states of the CFSM $\mathsf{G}\upharpoonright_{\mathsf{A}}$ to pairs $(\mathsf{A}, \tau) \in \mathcal{P} \times \mathbb{T}(\mathsf{G})$

[2]Our algorithm relies on abstract syntax trees of g-choreographies, but it does not depend on the precedence relation chosen.

Definition 3.8 (Realizing run)**.** A run $\pi \in \mathcal{R}(S)$ is $\Omega_{\mathsf{G},\lfloor}$*-realizing for* $\tau \in \mathbb{T}(\mathsf{G})$ if there exists a stable configuration s in π such that, for each $\mathsf{A} \in \mathcal{P}$, $q \in \Omega_{\mathsf{G},\lfloor}(\mathsf{A}, \tau)$ for the local state q of A in s.

Definition 3.9 (Oracle scheme)**.** A map $\Omega_{\mathsf{G},\lfloor}$ is an *oracle scheme of* G for $\_\lfloor\_$ if $WF(\mathsf{G})$ then for every $\tau \in \mathbb{T}(\mathsf{G})$ each maximal run $\pi \in \mathcal{R}(S)$ is $\Omega_{\mathsf{G},\lfloor}$-realizing for $\tau \in \mathbb{T}(\mathsf{G})$.

The main purpose of an oracle scheme Ω_{G} is to identify, for each point of the choreography (namely, for each syntactic subtree of G) a set of "expected" configurations that the system should intersect along its (maximal) executions.

Example 3.6. *Below is a (partially specified) possible oracle scheme for the g-choreography from Chapter 2.1.2 and the CFSMs shown in Figure 2.6:*

$$\Omega_{\mathsf{G},\lfloor}(\mathsf{A},\mathsf{G}) = \{q_e\} \qquad \Omega_{\mathsf{G},\lfloor}(\mathsf{B},\mathsf{G}) = \{q_e\} \qquad \textit{and} \qquad \Omega_{\mathsf{G},\lfloor}(\mathsf{C},\mathsf{G}) = \{q_e\}$$

where q_e denotes the state with the dangling outgoing transition in the corresponding machine. Basically, for the whole g-choreography G, the oracle scheme $\Omega_{\mathsf{G},\lfloor}$ yields the "last" states of the CFSMs. For the subtree $\tau = \mathsf{B}{\to}\mathsf{A}\colon \texttt{allow}$ we can define:

$$\Omega_{\mathsf{G},\lfloor}(\mathsf{A},\tau) = \{1,\ldots,5\} \qquad \Omega_{\mathsf{G},\lfloor}(\mathsf{B},\tau) = \{q_e\} \qquad \textit{and} \qquad \Omega_{\mathsf{G},\lfloor}(\mathsf{C},\tau) = \{q_e\}$$

which basically takes the first state that allows A to receive the message `allow` *or to choose an alternative branch.*

Test cases are then built by combining machines obtained by the split function and by identifying the success states via the oracle function, i.e., states that correspond to the execution of the interactions of the subtrees of the g-choreography:

$$\mathit{tests}(\mathsf{G},\mathsf{A}) = \{(\langle M_{\mathsf{B}}, \Omega_{\mathsf{G},\lfloor}(\mathsf{B},\tau)\rangle)_{\mathsf{B}\neq\mathsf{A}\in\mathcal{P}} \mid \tau \in \mathbb{T}(\mathsf{G}) \wedge \forall \mathsf{B} \neq \mathsf{A} \in \mathcal{P} :\ M_{\mathsf{B}} \in \mathsf{split}(\mathsf{G}\lfloor_{\mathsf{B}})\} \tag{3.2}$$

More intuitively, for every participant we select a single machine from the ones generated by split and combine them (exhaustively) into test cases. Each test case corresponds to a unique path of execution (i.e., selection of internal choices) of the original g-choreography.

To show that our test generation scheme is correct, we need to prove that, given a g-choreography G and a compatible well-formedness condition $WF(\mathsf{G})$, the test cases generated by $\mathit{tests}(\mathsf{G},\mathsf{A})$ are (G,A) – *suitable*.

Lemma 3.10. *Let G be a g-choreography, $\_\lfloor\_$ be a projection operation, and $T \in \mathit{tests}(\mathsf{G},\mathsf{A})$. Then $\mathcal{R}(\mathsf{G}\lfloor_{\mathsf{A}} \otimes T) \subseteq \mathcal{R}(\mathsf{G}\lfloor)$.*

Proof. Let $S = \mathsf{G}\restriction_{\mathsf{A}} \otimes T$ and $\pi = \{(s_i, l_i, s_{i+1})\}_{0 \leq i < n} \in \mathcal{R}(S)$. Note that the initial configurations s_0 of S and $\mathsf{G}\downarrow$ coincide, because the machines in the test have the same initial state of the machines in the projections for every participant. The proof is by induction on the length n of the run π.

When $n = 0$ the empty run belongs to any system, so the statement trivially holds. For the inductive step, let us assume that the thesis holds for all π with length k where $0 \leq k < n$.

Let $\pi' = \{(s_j, l_j, s_{j+1})\}_{0 \leq j < n-1}$ consisting of the first $n-1$ transitions of π. By the inductive hypothesis $\pi' \in \mathcal{R}(\mathsf{G}\downarrow)$.

Let X be the subject of l_n. We distinguish two cases.

- If $\mathsf{X} \neq \mathsf{A}$, let M_{X} be the machine for X in T, q be the local state of X in s_n, and q' the local state of s_{n+1}. By definition, $\mathsf{q} \xrightarrow{l_n} \mathsf{q}'$ is a transition in M_{X}. Hence, by definition of test $\mathsf{q} \xrightarrow{l_n} \mathsf{q}'$ is also a transition of $\mathsf{G}\downarrow_{\mathsf{X}}$. Hence, s_n has a transition with label l_n to s_{n+1} in $\mathsf{G}\downarrow$.
- Otherwise, if $\mathsf{X} = \mathsf{A}$, the machine for X is exactly the same as in $\mathsf{G}\downarrow$, so the last transition of π occurs in both systems.

□

Theorem 3.11. *If* $\mathit{WF}(\mathsf{G})$ *then every test case in* $\mathit{tests}(\mathsf{G}, \mathsf{A})$ *is* $(\mathsf{G}, \mathsf{A}) - \mathit{suitable}$.

Proof. Recall that $\_\downarrow\_$ is a projection operation compatible with WF. Let $S = \mathsf{G}\restriction_{\mathsf{A}} \otimes T$ and $T = \{(M_{\mathsf{X}}, \Omega_{\mathsf{G},\downarrow}(\mathsf{X}, \tau))\}_{\mathsf{X} \neq \mathsf{A}}$ for some τ subtree of G. We have to show that all finite maximal runs $\pi = \{(s_i, l_i, s_{i+1})\}_{0 \leq i < n} \in \mathcal{R}(S)$ contain a stable configuration s such that for every $\langle M_{\mathsf{X}}, \Omega_{\mathsf{G},\downarrow}(\mathsf{X}, \tau)\rangle$ in T $s \in \Omega_{\mathsf{G},\downarrow}(\mathsf{X}, \tau)$.

By Lemma 3.10 $\pi \in \mathcal{R}(\mathsf{G}\downarrow)$. In the following we let $s[\mathsf{X}]$ denote the local state of the machine of participant X in the configuration s. We distinguish two cases:

- if π is maximal in $\mathsf{G}\downarrow$ then, by definition of oracle scheme, there is a stable configuration s in π such that $s[\mathsf{X}] \in \Omega_{\mathsf{G},\downarrow}(\mathsf{X}, \tau)$ for all $\mathsf{X} \neq \mathsf{A}$, which concludes the proof
- otherwise, there is a transition $s_{n+1} \xrightarrow{l} s$ in $\mathsf{G}\downarrow$. Note that $\mathsf{X} = \mathsf{sbj}(l) \neq \mathsf{A}$ since otherwise such transition would be possible also in $\mathsf{G}\restriction_{\mathsf{A}} \otimes T$ because $\mathsf{G}\restriction_{\mathsf{A}}$ would

have an enabled transition from its local state $s_{n+1}[\mathsf{A}]$; this would violate the hypothesis that π is maximal in $\mathsf{G}\upharpoonright_{\mathsf{A}} \otimes T$. Also,

$$s_{n+1}[\mathsf{X}] \xrightarrow{l} \mathsf{q} \in \mathsf{G}\upharpoonright_{\mathsf{X}} \qquad \text{and} \qquad s_{n+1}[\mathsf{X}] \xrightarrow{l} \mathsf{q} \notin M_{\mathsf{X}}$$

where the first relation must hold by the definition of the semantics of communicating systems and the second relation holds, otherwise the maximality of π would be violated. We proceed by contradiction; there are two cases:

- l **is an output:** by definition of split, $s_{n+1}[\mathsf{X}] \xrightarrow{\mathsf{X}\,\mathsf{Y}!m} \mathsf{q}'$ in M_{X} where $\mathsf{X}\,\mathsf{Y}!m \neq l$. This yields a contradiction because the machine M_{X} would have an enabled output from the configuration s_{n+1}, contrary to the hypothesis that s_{n+1} is maximal in S.
- l **is an input:** we have $s_{n+1}[\mathsf{X}] \xrightarrow{l} \mathsf{q}'$ in M_{X} with $\mathsf{q}' \neq \mathsf{q}$, since the split operation would remove an input transition from $s_{n+1}[\mathsf{X}]$ only if $s_{n+1}[\mathsf{X}]$ has another input transition with the same label. Then we have a contradiction since $s_{n+1}[\mathsf{X}] \xrightarrow{l} \mathsf{q}'$ is enabled in s_{n+1}, which again violates the maximality of π in S.

In both cases we have a contradiction.

□

3.3 Tooling

In this section, we present ChorTest[3], an implementation of our framework extending ChorGram, a toolchain for the specification and analysis of g-choreographies.

A brief overview of the architecture of ChorTest is illustrated in Figure 3.1. Users interact with ChorTest through a command line interface implemented in Python, with several subcommands exposing the different functionalities of the tool. The tool relies on GraphViz [103] to render raster graphics of the diagrams produced by the toolchain (i.e., g-choreographies and CFSMs).

The input of ChorTest consists of a g-choreography modelling the communication protocol for which tests are desired in the form of a .gc file — a text file carrying the g-choreography (cf. [104]). Through the project command, ChorTest invokes ChorGram's gc2fsa feature to obtain a set of CFSMs representing the local views of the g-choreography in input.

[3] Available at `https://bitbucket.org/eMgssi/chorgram/` as part of ChorGram and at `https://github.com/alxrcs/chortest` as a standalone tool.

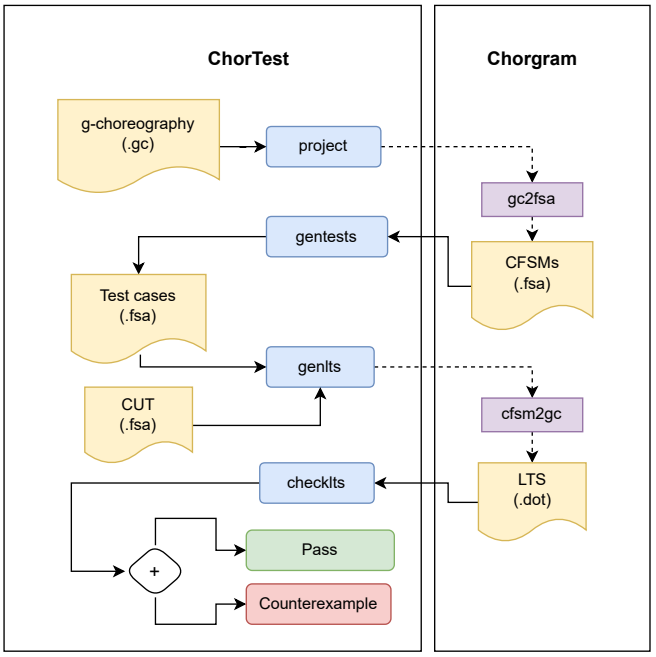


FIGURE 3.1: The architecture of ChorTest (simplified)

The CFSMs returned by gc2fsa are stored in a set of files used to build proper cases by means of the gentests command; these files are in fsa format for ChorGram's internal representation of CFSMs. This command of ChorTest implements the split operation that generates valid test cases, as described in Chapter 3.2. These test cases can then be used to test a specific realization of one of the components of the system, using ChorTest's genIts command. This command accepts a specific test case (as generated by the gentests command) and optionally a .fsa file modelling the component that the user wishes to test. The user can also check for compliance as defined in Definition 3.6 by using the checkIts command, storing all resulting information in a log file.

Since ChorTest instantiates our framework with the projection operation from [27], we can use a simple oracle scheme to signal successful executions. Although more complex oracles can be specified, the default oracle scheme we use in ChorTest simply marks as accepting states those which do not have outgoing transitions in each test CFSM. Due to the way in which our projection operation works, intuitively, these states correspond to the execution of the whole choreography.

To deal with g-choreographies with loops, we also provide an unfold subcommand which unfolds the loops in the g-choreography in input, producing a new g-choreography without loops. As described in Chapter2, this command produces a finite unfolding of the g-choreography, which can then be directly used to generate test cases, or even modified so that the flow of the execution follows a particular execution path (e.g., a path that the developer might consider interesting). Since unfolding the loops results

in simple choreographies that can be handled sequentially, the test cases generated by gentests are still valid, For example, when a choreography contains a loop with a choice, this will result in the choice being unfolded multiple times, and subsequently all possible interleavings being generated as tests.

The gentests functionality does not faithfully follow the definition of the split operation given in Chapter 3.2. In fact, for the sake of clarity, the definition of the split(M) function takes the union over all the non-deterministic states of M. This results in a high number of redundant tests. In fact, we can use the following theorem to optimize our algorithm. Theorem 3.12 states indeed that the split operation is insensitive to which non-deterministic state of M is chosen to generate the tests at each step.

Theorem 3.12. *Let* $q, q' \in nds(M) : q \neq q'$. *Then* $split(M, q) = split(M, q')$.

Proof. The proof by induction on the (set of) edges of M. The base case is trivial, and is when $nds(M) = \emptyset$.

For the inductive case, let $q, q' \in nds(M) : q \neq q'$ and $M' \in \mathsf{split}(M, q)$. We consider only the cases where both q and q' have outputs, since the other cases are similar. We have:

$$\begin{aligned} \mathsf{split}(M,q) = & \bigcup_{q \xrightarrow{\mathsf{A\,B!m}} q_1} \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} q_1) && \text{by definition} \\ \implies & M' \in \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q}) && \text{for some } q \xrightarrow{\mathsf{A\,B!m}} \bar{q} \end{aligned}$$

Notice that

- $q' \in nds(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q})$ by construction, and
- by induction, for every $q'' \in nds(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q})$, we have

$$\mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q}, q') = \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q}, q'')$$

Hence, $M' \in \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q}, q')$.

$$\begin{aligned} & \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q}, q') = && \text{by definition} \\ & \bigcup_{q' \xrightarrow{\mathsf{A\,B!m}} q'_1} \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q} - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} q'_1) \implies \\ & M' \in \mathsf{split}(M - M(q) + q \xrightarrow{\mathsf{A\,B!m}} \bar{q} - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} \bar{q}') && \text{for some } q' \xrightarrow{\mathsf{A\,B!m}} \bar{q}' \end{aligned}$$

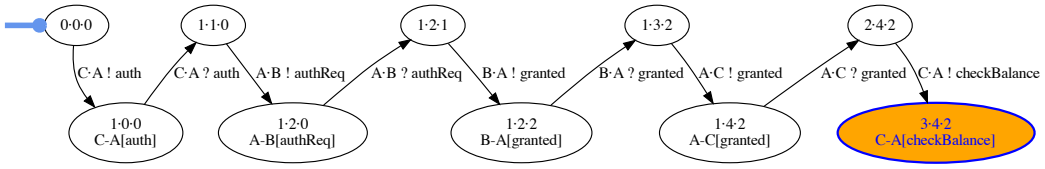


FIGURE 3.2: A visual representation of a deadlocked LTS

We now show that $M' \in \mathsf{split}(M, q')$.

$$\begin{aligned}
\mathsf{split}(M, q') = & \bigcup_{q' \xrightarrow{\mathsf{A\,B!m}} q_1'} \mathsf{split}(M - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} q_1') && \text{by definition} \\
\implies & \mathsf{split}(M, q') \supseteq \mathsf{split}(M - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} \vec{q}') \\
\supseteq & \mathsf{split}(M - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} \vec{q}', q) \\
= & \bigcup_{q \xrightarrow{\mathsf{A\,B!m}} q_1} \mathsf{split}(M - M(q') + q' \xrightarrow{\mathsf{A\,B!m}} \vec{q}' - M(q) + q \xrightarrow{\mathsf{A\,B!m}} q_1) && \text{by definition.}
\end{aligned}$$

This concludes the proof. □

The output of ChorTest consists of a tagging of tests as passed or failed. In the latter case, the labelled transition system (LTS) of the test execution (i.e., the LTS resulting from the combination of the test case and the given CFSM) can be used to identify problems in the realization at hand. This is done by exploiting ChorGram's feature to export these LTSs in dot format. For instance, ChorGram marks deadlock configurations of LTSs in the dot representation. The visual representation rendered by GraphViz provides feedback about problematic executions. An example of such representation for a failed test is one of cases in Figure 3.2, where the filled configuration is a deadlock[4]. This should help the user fix errors, and we plan to improve this feature to make it more useful when the LTSs are large.

3.4 The ATM Protocol: a case study

We consider the g-choreography in Figure 3.3 involving the participants A, B, and C, i.e. respectively the ATM, the bank, and a client as in Figure 1.1. Observe that the running example from Figure 1.1 is a sub-choreography of the g-choreography in Figure 3.3.

[4]Notice that the message `checkBalance` is left in the buffer from C to A in the filled configuration.

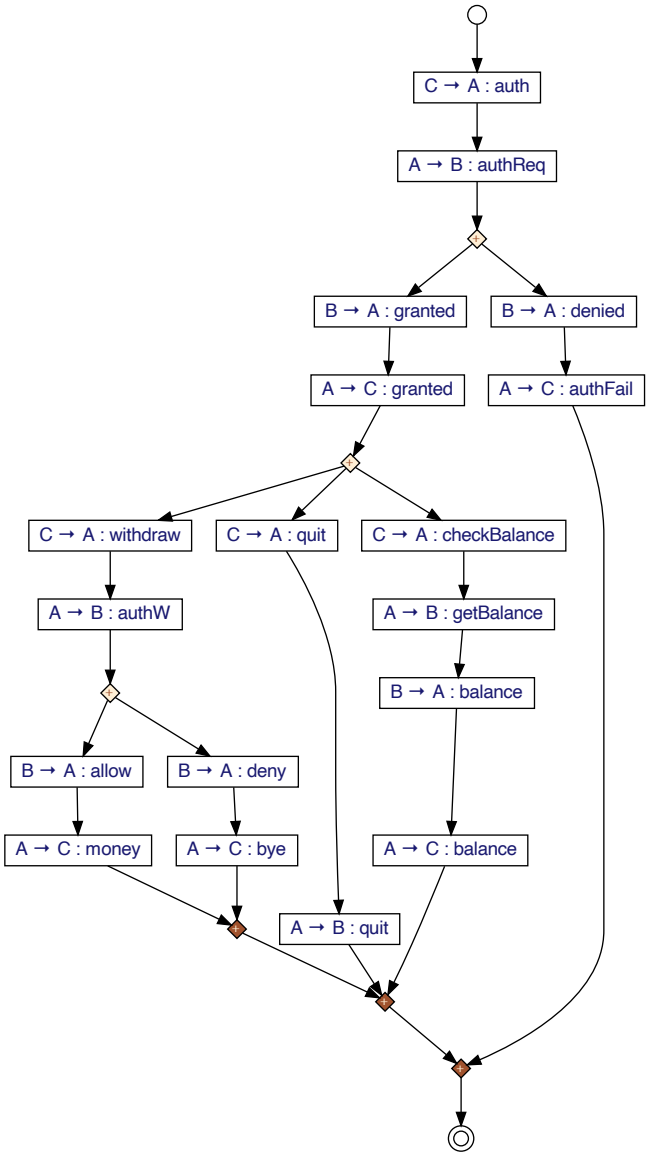


FIGURE 3.3: The complete choreography for the ATM scenario

The client starts a session of the protocol by authenticating with the ATM (`auth`). The ATM then delegates the authentication to the bank, which can either reject or accept the request by replying with either a `denied` or a `granted` message. In both cases the ATM forwards the authentication result to the client. The choreography terminates if the authentication fails. If the authentication succeeds then the ATM offers three options to the client: (M) withdraw money (`money`), (Q) terminate the session (`quit`), or (C) check the account balance `checkBalance`.

In case (C), the ATM requests to the bank the balance and forwards the result to the client via a `balance` message. In case (Q), the ATM simply notifies the bank of the termination of the session. Case (M) is the choreography of Figure 1.1 whereby the withdrawal request is forwarded to the bank which decides whether to allow or deny the request.

Let the ATM A be our CUT and M_{A} be a possible implementation A to be tested. We propose to use a CFSM for B and one for C, say M_{B} and M_{C} respectively, paired up with a subset of their states, say Q_{B} and Q_{C} respectively. These sets are used to capture legit executions of the test, and are the states where the test machines should be after proper message exchanges according to the protocol.

Not all machines are good candidates for tests. For instance, using the machines of Figure 3.4 to test A would introduce uncertainty (a possible source of *flakiness* [105–107]) in determining the outcome of the test, since the non-deterministic behaviour of B and C could produce different executions in different runs. Therefore, as seen previously, we require test machines to consist of deterministic machines only. Running the test T on M_{A} would correspond to executing a communicating system S, consisting of the three CFSMs M_{A}, M_{B}, and M_{C}. The CFSM M_{A} *fails* T when S has at least a run that has no "acceptable" configuration. By acceptable configuration, we mean a configuration where messages have been properly exchanged and both M_{B} and M_{C} are in a local state belonging to Q_{B} and Q_{C} respectively. If that does not happen then M_{A} *passes* the test.

Intuitively, this amounts to say that M_{A} fails the test when it has a run with T where either the messages are not exchanged according to the protocol or the test machines are led to "unexpected" states after some interactions.

In principle, CFSMs M_{B} and M_{C} could be attained in many ways. In a choreographic framework, though, there is a natural way to attain test machines: one can use projections. The intuition is that, on a well-formed g-choreography, projections yield realizations of non-CUT components that are correct *by construction*. In fact, our algorithm operates on the projections of the non-CUT participants. In the ATM example, using

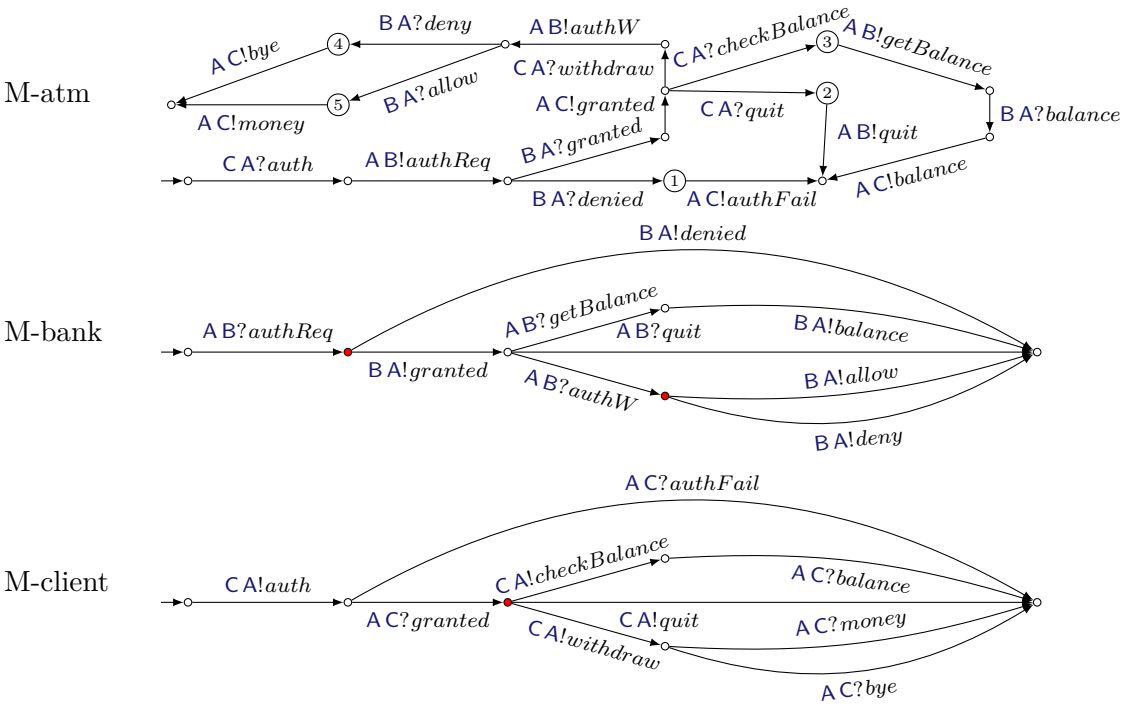


FIGURE 3.4: Projections of the choreography of Figure 3.3

again the projection operation[5] in Chapter 2.2, we obtain the three CFSMs of Figure 3.4, where some states are coloured for illustrative purposes as described further. For each non-CUT machine $M \in \{\text{M-bank}, \text{M-client}\}$, our algorithm generates tests out of each; the algorithm firstly enumerates the states, that is, the states in which M can disregard some possible computations and progress independently of other machines: in Figure 3.4, the non-deterministic states are the states filled with red.

The second phase of our generation algorithm *splits* non-CUT machines at their non-deterministic states. Intuitively, this corresponds to letting each non-CUT machine follow only one of its possible non-deterministic branches. For instance, the splitting operation on the machines M-bank and M-client of Figure 3.4 yields the CFSMs in Figure 3.5 where, for the sake of conciseness, we remove unreachable states and omit isomorphic CFSMs. We also mark some states with outgoing dangling transitions; as we will see later, these states are used to set the success criteria to pass tests, by choosing states in the test machines that correspond to specific points in the g-choreography. For the moment, it is enough to assume that states with an outgoing dangling transition correspond to the execution of the whole choreography.

How can one choose the set Q_{B} and Q_{C} to form the test T? This is not an easy endeavour in general; for instance, choosing sets that are bluntly "unrelated" to the g-choreography

[5] We let the CFSM M-atm in Figure 3.4 be non-minimal, as collapsing the two states without outgoing transitions would have compromised readability. We also numbered some of its states, to refer to them later on.

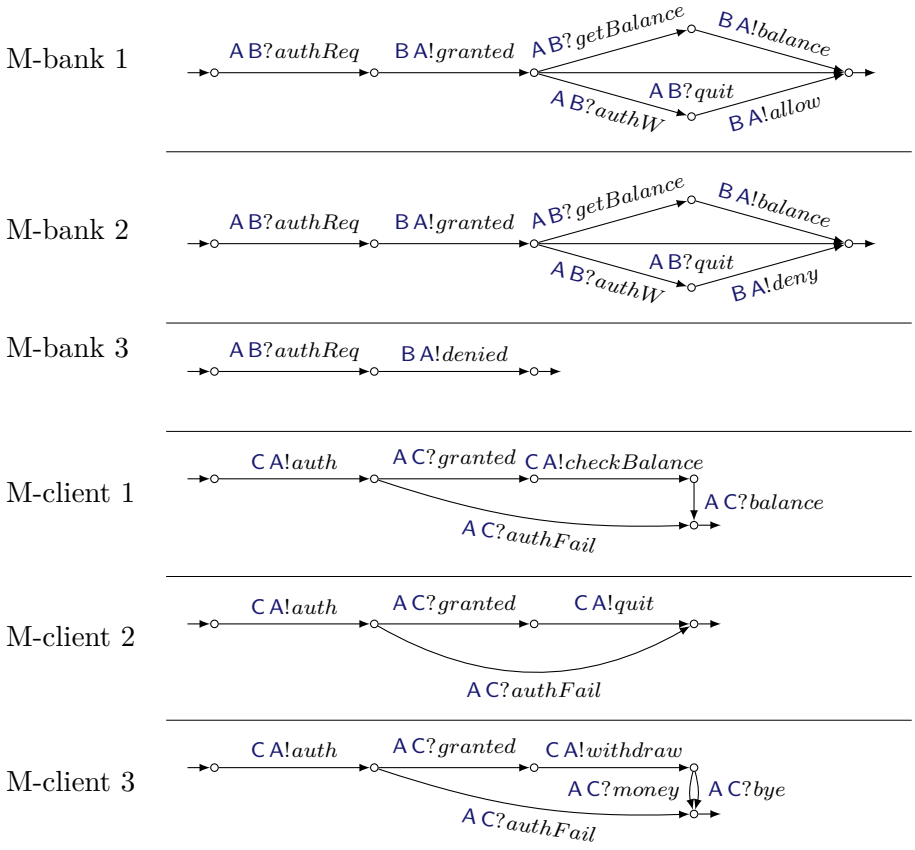


FIGURE 3.5: CFSMs resulting from splitting the projected CFSMs for B and C

could yield tests that are impossible to pass even for correct realizations. This is related to the so-called *oracle problem* discussed in Chapter 3.2. We propose a criterion to tackle the oracle problem again using the g-choreography. As discussed in Chapter 3.4, natural candidates for such criteria are "points" in the g-choreography corresponding to valid executions of the protocol. These points correspond to some states of the machines. For instance, a correct execution of the protocol once the client asks to check the balance when the g-choreography in Figure 3.3 reaches the interaction C→A: `checkBalance`. The states corresponding to this point in the tests are respectively, (cf. Figure 3.4) the starting state of transition A B?*getBalance* for M-bank and the target state of transition C A!*checkBalance* for M-client.

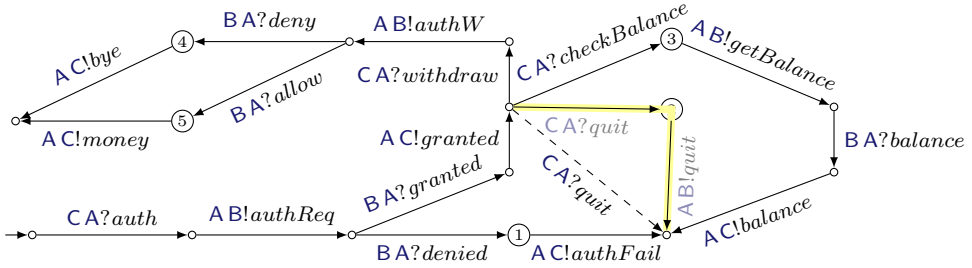
Finally, tests are obtained by combining in all possible ways each split machine of non-CUT machines. For instance, in our example this yields nine tests which are the pairs (M-bank i, M-client j) for $i, j \in \{1, 2, 3\}$ formed out of the machines for the bank and client from Figure 3.5.

We now return to the ATM example to illustrate how test cases generated by our algorithm can flag faults. The way in which we devise these examples is somewhat analogous to what is known as mutation testing in traditional software testing. Recall that the

main goal of mutation testing is ensuring the quality of test cases by introducing changes in the source code. In our case, we create faulty realizations of the ATM, by altering the CFSM for A, obtained by projection from the g-choreography in Figure 3.3. For the sake of representation, we will visually remark similarities and differences between faulty realizations and the correct ones (that is, the projected ones).

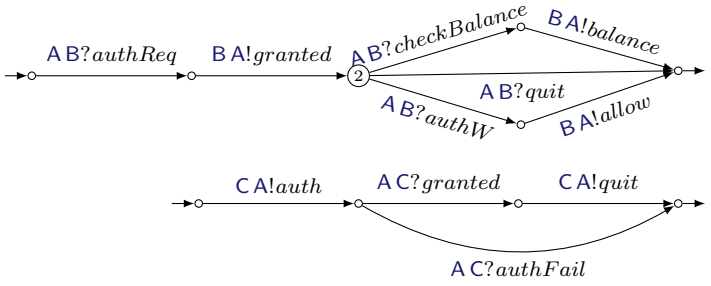
Example A.1 We consider a realization that is similar to the correct one shown in Figure 3.4, except that it does not notify the bank when the client quits the protocol. This behaviour is reflected by the following CFSM:

A.1 =



where the highlighted path of the original machine is replaced by the dashed transition. Note that this realization of the ATM never sends `quit` to the bank as wanted.

One of the test cases generated by our algorithm is

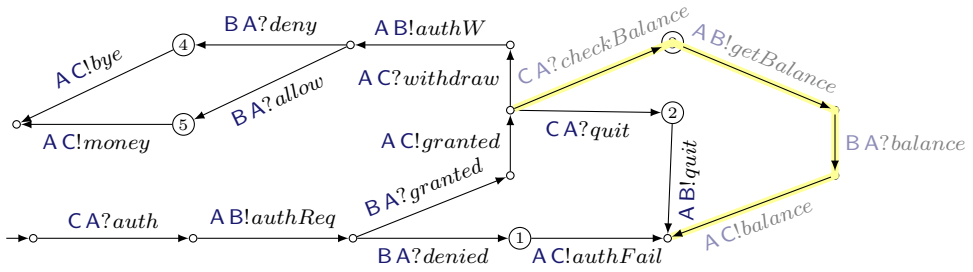


consisting of the machines M-bank 1 and M-client 2 from Figure 3.5. We expect that the action A B?*quit* in the bank machine is never executed when running in parallel with A.1 since the latter does not execute the output A B!*quit*. Hence, the bank machine never reaches a state belonging to the oracle (marked with dangling arrows in these examples), remaining stuck in state 2.

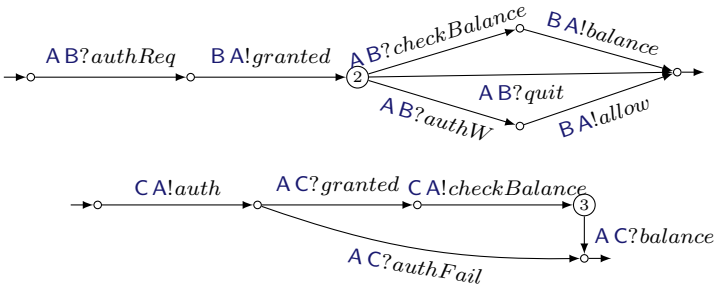
Example A.2 Our next example consists of an ATM which is not able to process the `checkBalance` message. This is analogous, for example, to cases where a specific feature

has not been yet implemented. That is:

A.2 =



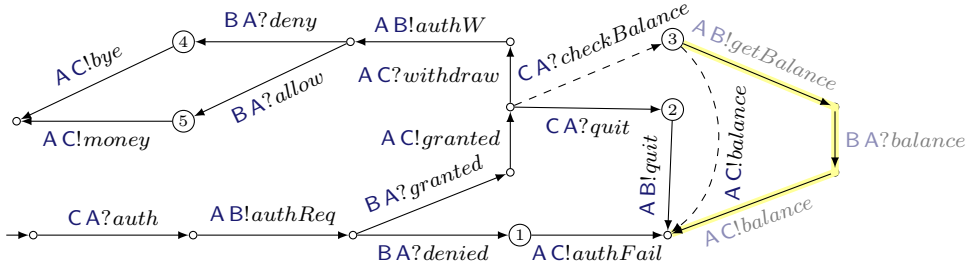
where the highlighted path represents the missing transitions (with respect to the projection of A). Any test in which the bank approves the authentication request and the client tries to check its balance will be able to detect this fault. In this case, we show the one composed of M-bank 1 and M-client 1, also from Figure 3.5.



In this case, note that the machines composing the test case would remain in states 2 and 3 respectively, not being able to reach a successful configuration.

Example A.3 Finally, let us consider a case where the ATM returns a (possibly incorrect) balance, without performing the expected interaction with the bank:

A.3 =



where the dashed path replaces the highlighted one in the projection of A. This fault can also be detected through the test case used for A.2, since the machine corresponding to the bank would also remain forever in state 2, due to the lack of further interaction from the ATM.

3.5 Evaluation

In this section, we report some experimental results obtained with ChorTest on the ATM scenario. More precisely, we consider the faulty realization presented in Section 3.4. For this scenario, the experiments have been conducted as follows:

1. project the well-formed choreography of the scenario into the set of CFSMs,
2. split each projection and generate all the test cases out of the split machines,
3. identify the CUT together with its test cases,
4. check for test compliance on each LTS corresponding to the execution of the CUT with its test cases.

We use the default oracle scheme from ChorTest, that is, we mark as accepting states those which do not have outgoing transitions in each test CFSM.

Table 3.1 summarizes the results obtained for the ATM scenario. All experiments were run on a 16 GB Intel Core i7-6700HQ machine, on Ubuntu 20.04 and Python 3.8.6.

	Tests				LTS Gen.		Compliance		
Example	Project	Gen.	CUT	# tests	Tot.	Avg.	Tot.	Avg.	Fail
Projs.	0.0095	0.3874	All	19 (89)	0.4423	0.0232	0.4235	0.0222	0
A.1	0.0096	0.3522	A	12 (72)	0.3062	0.0255	0.2722	0.0226	2
A.2	0.0102	0.3379	A	12 (72)	0.2979	0.0248	0.2563	0.0213	2
A.3	0.0094	0.3333	A	12 (72)	0.2992	0.0249	0.2868	0.0239	2

TABLE 3.1: Results for the ATM scenario (all times are in seconds)

In each table, the first column refers to both the correct projections (for the first row) or the variants of the projections discussed in 3.4 (the remaining rows). Column "Tests" yields the time it takes to project each g-choreography, as well as the time it takes to generate tests; recall that the first operation is done by ChorGram while the generation is performed by ChorTest. Also, the latter operation includes both the splitting and combination of the CFSMs to derive the test cases. Note that column "Tests" corresponds to the first two phases of the experiments. It is worth remarking that these operations are done only once and do not depend on the CUT.

In the third phase of the experiments, relevant tests are identified depending on the selected CUT. For the projections (first row) we consider all the possible combinations, while, in each of the other experiments, we test a specific faulty component (as shown in the previous sections) and report the number of tests generated for it in the column "# tests". The numbers in round brackets refer to the non-optimized variant of ChorTest.

Columns "LTS Gen." and "Compliance" correspond to the final phase of the experiments and report the time ChorGram takes to generate the LTS of the test case in parallel with the CUT and the time spent by ChorTest on checking for compliance. Since (unlike the projection and generation stages) these last steps are done once per each generated test, we report both the total and the average times.

Although more experiments would be required, we can draw some preliminary conclusions.

Firstly, note that the times for the projection and generation are mostly identical, since these first stages execute the exact same operations for each experiment. Also, projecting g-choreographies is a relatively inexpensive operation when compared to generating tests, computing LTS, and checking for compliance. This is also due to the fact that the number of tests generated depends directly on the number of internal choices that components (other than the CUT) possess.

For the ATM examples, all operations have comparatively similar numbers, due in part to the small size of the example and the lack of parallel composition [6].

Theorem 3.12 guarantees that our optimization does not remove any relevant tests, and at the same time enables for rather large time savings, especially in those examples with parallel composition. Firstly, the speed-up obtained from our optimization is directly related to the considerable reduced number of tests cases generated.

Another common way of assessing the quality of a test suite is through *mutation testing*, a technique commonly applied by injecting faults into code or a model, and then checking whether the given test suite can detect the faults. This modified version of the code or model is called a *mutant*[7].

The goal of mutation testing is to determine whether a test suite is able to detect (and thus "kill") the mutants. If some test case fails when executed on a mutant, then the mutant is said to be *killed*; otherwise, the mutant is said to have *survived*. The metrics obtained from this process can be used to characterize the quality of a test suite, e.g., the *mutation score*, which is the ratio of the number of killed mutants to the total number of mutants.

Multiple mutants can be generated by changing a model in different ways. The most common way of generating mutants is by applying a set of *mutation operators* to the model. A mutation operator is a transformation that changes a model in a specific

[6] Parallel composition tends to greatly increase the size of the LTS, because all possible interleavings must be accounted for

[7] Since our framework considers model-based testing at an abstract level, we'll talk about mutation testing in the context of models exclusively.

way. These operators depend mostly on the application domain (e.g., the programming language or the type of model being used). For instance, in the context of Java, a mutation operator could change the value of a variable, or remove a statement from the code. In the context of UML models, a mutation operator could change the type of a class, or remove an association between two classes. In our framework, the mutation operators we define are presented in 3.2.

It is worth noting that, since the mutation operators must introduce faults, a general prerequisite for their application is that their changes must involve the component under test in some way. That is, the CUT's behaviour must change in a way that is (ideally) observable by the test suite. This translates to a few simple conditions: Mutations that apply directly to interactions require the CUT to be either the sender or the receiver of the message. In the case of the RMO operator, the CUT should be the destination for the output (note that it cannot be the input, since mutation operators are not allowed to change the CUT's behaviour). We also made sure that the RMO operator was only applied in cases where the CUT had no other possible action to perform, so that the mutation would be observable.

Table 3.2 presents the operators we designed and applied. At the local level, mutation operators are applied systematically directly to the local model of the participant under test. In our case, this involves modifying the CFSM corresponding to the projection of the CUT. After applying a specific mutation operator, we compose the resulting local model with the test suite and execute the resulting system.

Operator	**Description**
Remove Output (RMO)	Removes an output from a CFSM.
Change message type (CML)	Changes the message type in an interaction to a different (wrong) one
Change interaction type (CIT)	Changes an interaction from an output to an input and vice versa
Remove State (RST)	Removes a state from a machine
Repeat Transition (RTR)	Repeats a transition by adding an intermediate state

Table 3.2: Local mutation operators

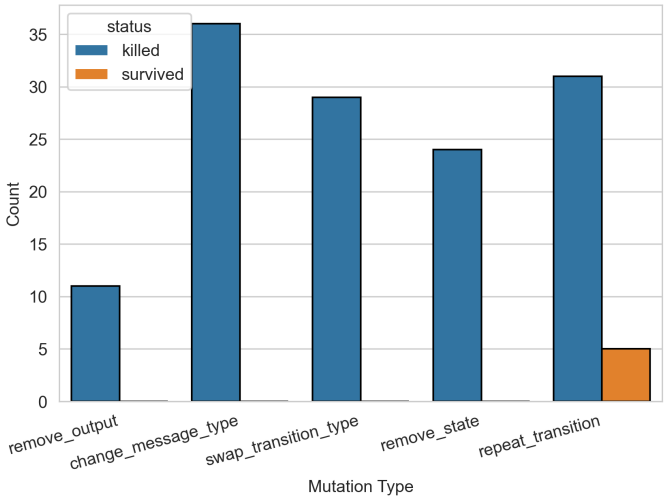


FIGURE 3.6: Mutation testing results for the ATM example

Figure 3.6 shows the results of applying mutation testing to the ATM example. The figure shows how many mutants resulted for each category (according to the type of mutation applied), as well as whether the mutant survived or was killed.

Figure 3.7 shows the CFSMs of participants, and the transitions that were affected by the RTR mutation operator, for the cases where the mutation survived.

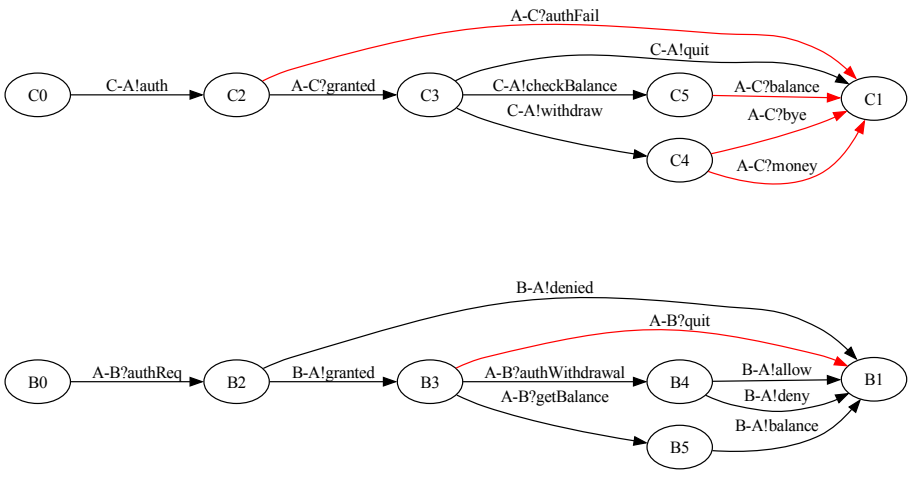


FIGURE 3.7: Projections for the ATM example with surviving mutants marked

The mutation testing results show that the test suite is able to detect all the mutants generated by the mutation operators, achieving an overall mutation score of 0.963. However, as shown in Figure 3.7, there are surviving mutants specifically due to the RTR

operator being applied to a "receive" transition that was the last one in the sequence. The mutation score for this particular operation is much lower, at 0.454. This indicates that the test machines are not able to detect this specific type of mutant, as they cannot observe whether the message was received or not by the CUT.

Figure 3.8 also shows the time it took to check each mutant, for each of the tests. Machine A took the longest to test, which is expected as it acts as an intermediary between the other participants, aside from having the most transitions.

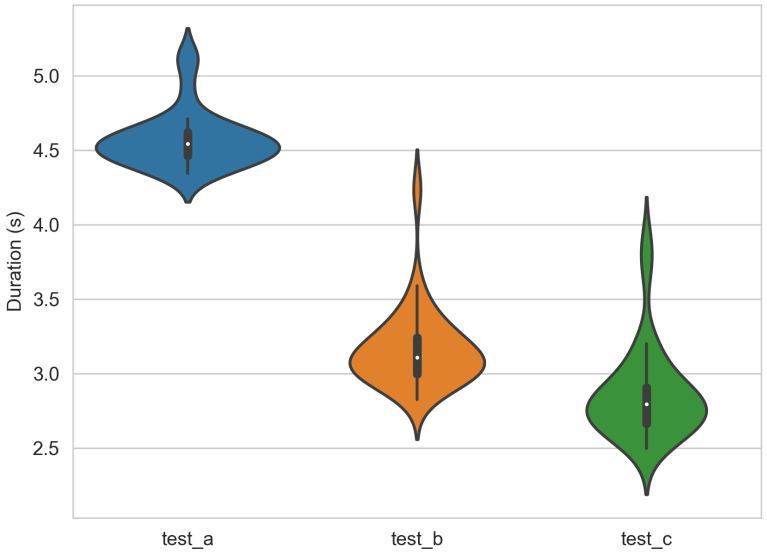


FIGURE 3.8: Time taken to check mutants for the ATM example for each test

Figure 3.9 shows the time it took to check each mutant, for each of the mutation operators. The figure shows that the RMO operator takes slightly longer than the rest, but the differences are not significant.

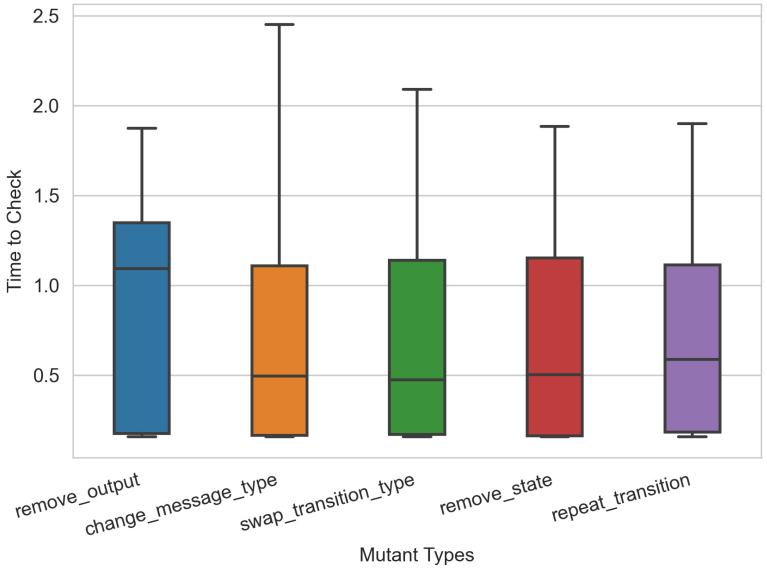


FIGURE 3.9: Time taken to check mutants for each type of mutation

Chapter 4

On the Synthesis of Executable Tests

4.1 Introduction

In this chapter, we present a mechanism for the synthesis of executable tests, based on the abstract framework discussed in previous chapters. The contributions of this chapter can be divided into two primary aspects:

Firstly, we introduce a collection of local session types (implemented in Python) that enable the representation of the abstract local protocols outlined in Chapter 3. These types also facilitate a degree of static verification. Secondly, we present a tool capable of generating executable Python tests derived from choreographic models, utilizing the aforementioned types as a foundation. These functionalities further extend the ChorTest toolchain introduced previously.

In general terms, the envisioned workflow proceeds as follows: Initially, the g-choreography is validated, including the assessment of well-branchedness and well-formedness conditions, by employing the abstract framework introduced in Chapter 3. Subsequently, the local session types are extracted in the form of Communicating Finite State Machines (CFSMs). These CFSMs are then utilized to generate executable tests tailored to a specific participant. Once created, these tests can be executed to evaluate the behaviour of individual system components, ensuring their adherence to the original communication protocol represented by the g-choreography.

4.2 Statically-Typed Local Session Types in Python

This section delves into the implementation of local session types in Python. One of our primary objectives was to maintain simplicity in the implementation while effectively representing the local session types outlined in Chapter 3. Additionally, we aimed to align the implementation closely with the original formalism, enabling readers to easily comprehend the types.

Though Python is renowned for its dynamic typing, support for static type annotations has been introduced since version 3.5. These annotations allow developers to specify the type of a variable or function parameter, as well as to conduct some degree of incremental static type checking using tools like MyPy [108] or Pyright [109]. At the time of writing, Pyright can be employed, for example, as a command-line tool or through the Pylance language server, integrated with the Visual Studio Code editor.

The type system we have developed takes inspiration from the one described by Jespersen et al. [110], which is, in turn, based on the Haskell library devised by Pucella and Tov [111]. Our approach extends the original type system to accommodate multiparty communication, thereby allowing the representation of the local session types described in Chapter 3.

At the heart of the type system lies the foundational `LocalProtocol` type, which represents a local session type. This type features a single attribute, `transport` of type `Transport` type. Other classes inherit from this base class to define their specific local protocols.

Let `M` range over message types, let `R`, `S` range over participants, and `P`, `LP`, `RP` range over protocols[1]. A `LocalProtocol` type can be one of the following:

- `Eps` — the empty type. It represents the end of a session, and only allows for a `close` action.
- `Send(R, M, P)` — allows for sending a message. The type parameter `M` represents the type of the message, `R` represents the type of the recipient, and `P` represents the continuation type.
- `Recv(S, M, P)` — allows for receiving a message. The type parameter `M` represents the type of the message, `S` represents the type of the sender, and `P` represents the continuation type.

[1] In Python code, this translates to generic types constrained by some bounds (e.g., bound=ParticipantType, bound=MessageType) to ensure that the types passed during instantiation adhere to certain constraints.

$$\frac{}{\Gamma \vdash \texttt{Eps} : \texttt{L}}\ (\textsf{Eps})$$

$$\frac{\Gamma \vdash P : \texttt{L}}{\Gamma \vdash \texttt{Send}(R, M, P) : \texttt{L}}\ (\textsf{Send})$$

$$\frac{\Gamma \vdash P : \texttt{L}}{\Gamma \vdash \texttt{Recv}(S, M, P) : \texttt{L}}\ (\textsf{Recv})$$

$$\frac{\Gamma \vdash LP : \texttt{L} \qquad \Gamma \vdash RP : \texttt{L}}{\Gamma \vdash \texttt{Choice}(R, M, LP, RP) : \texttt{L}}\ (\textsf{Choice})$$

$$\frac{\Gamma \vdash LP : \texttt{L} \qquad \Gamma \vdash RP : \texttt{L}}{\Gamma \vdash \texttt{Offer}(S, M, LP, RP) : \texttt{L}}\ (\textsf{Offer})$$

FIGURE 4.1: Simple typing rules for the `LocalProtocol` (abbreviated as type `L`).

- `Choice(R, M, LP, RP)` — allows the current participant to make an active choice in the protocol. The type parameter `M` represents the type of the message, `R` represents the type of the recipient, and `LP` and `RP` represent the continuation types for the two branches of the choice. The developer can pick either branch and the protocol continues accordingly.
- `Offer(S, M, LP, RP)` — allows the current participant to choose a continuation depending on the type of message `M` and the identity of the sender `S` (i.e., a passive choice). The type parameter `M` represents the type of the message, `S` represents the type of the sender, and `LP` and `RP` represent the continuation types for the two branches of the choice. The protocol continues at runtime depending on the type of the message received.

With these building blocks, recursion can be modelled by simply defining a type alias[2] and using it as the continuation types `P`, `LP`, or `RP`. In our examples, however, being consistent with our assumptions from Chapter 3, we assume only finite unfoldings of the loops.

Over the course of an execution, the same instance of the `LocalProtocol` class will change between these classes, and after each action which has a continuation type, the protocol is updated to the continuation type. For example, if we have a protocol `P` of type `Send(R, M, P)`, then after sending a message of type `M` to the recipient `R`, the protocol is updated to `P`. This is accomplished by changing the `__class__` attribute of the protocol

[2]In Python, a type alias is defined by simply assigning the desired type to the desired name. That is, in the statement `A = int`, `A` becomes a type alias for the type `int`.

```
1          ReqRepClient = Send[Server, Request, Recv[Server, Reply, Eps]]
2          ReqRepServer = Recv[Client, Request, Send[Client, Reply, Eps]]
```

Listing 1: Python Local Type Definitions for a Simple Request-Reply Protocol

```
1      EvenClient = Send[Server, number,
2                        Offer[Server, No, Eps,
3                              Recv[Server, Yes, Eps]
4                        ]
5                  ]
6
7      EvenServer = Recv[Client, number,
8                        Choose[Client, No, Eps,
9                               Send[Client, Yes, Eps]
10                       ]
11                 ]
```

Listing 2: Python Local Type Definitions for a Protocol Involving a Choice

instance at runtime[3]. We also recursively update the generic type parameters for the continuation protocols.

These classes can be combined to create more complex communication protocols between participants, ensuring type safety and readability of the code.

Example 4.1. *To illustrate the use of the* `Send` *and* `Recv` *types, the code shown in Listing 1 shows the Python types for a simple Request-Reply protocol, where a server should reply to the client once it receives a request from it. The* `ReqRepClient` *and* `ReqRepServer` *types represent the protocol for the server and client respectively, and the* `Request` *and* `Reply` *types are simple types, possibly text.*

Example 4.2. *The code shown in Listing 2 shows the Python types for a protocol where the server should reply to the client with either a* `Yes` *or* `No` *message type, indicating whether the number it received is even or not. This example illustrates the use of the* `Choice` *and* `Offer` *types to convey branching in the protocol. Note that the leaves of the tree (lines 3 and 9) should be of type* `Recv` *(or* `Send`*) instead of* `Offer` *(or* `Choice`*).*

4.2.1 The Transport Layer

In order to actually exchange messages, every instance of a `LocalProtocol` subtype (i.e., `Send`, `Recv`, etc.) has an attribute of type `Transport`, which abstracts away the medium used to send and receive messages. In essence, this is used to implement the

[3]Changing an object's class at runtime is possible in Python, but only for classes that have a similar memory layout, i.e., classes having the same member attributes. In our case this is fine, since all subtypes of `LocalProtocol` have a single attribute of type `Transport`.

`send` and `recv` actions. The `send` and `recv` methods are asynchronous, and return a `Future` object which can be used to wait for the result of the operation.

An additional `peek` action is also required, which allows checking whether a message is available without actually receiving it. At the moment, this is required for the implementation of the `Offer` type, which needs to check whether a message is available before deciding which branch to take, since n-ary choices are modelled as a sequence of binary choices. This is necessary to avoid blocking the protocol execution when a message arrives for which the action is deeply nested in the offer subtree.

The interface of the `Transport` class is shown in Listing 3. To avoid accidental instantiation of the class, it is marked as abstract[4], by using the `ABC` class as a superclass (line 1). For this reason, the `Transport` class should be inherited from when implementing a specific transport layer (e.g., sockets, IPC, RPC, etc.). The `Transport` class is also not meant to be used directly by the developer, but rather through the `LocalProtocol` class, which provides a higher-level interface to the transport layer. For this reason, the `Transport` class does not have any generic type parameters nor does any complex type-checking.

In our implementation, we use the `asyncio` library to implement the transport layer. The `asyncio` library is part of the standard library in Python 3.7 and later, and is available as a third-party library for earlier versions. It offers support for asynchronous programming, which allows writing code that can run concurrently with other code through the use of the `async` and `await` keywords, without the need for explicit thread management. This is particularly useful for implementing the transport layer, responsible for sending and receiving messages, and which can be implemented in a non-blocking way given that a message loop is already running.

We provide an implementation of the `Transport` class that uses the `asyncio.Queue` class to implement the pairwise message queues between participants described in Chapter 3. It is worth noting that this implementation is not meant to be used in production, but rather as a proof-of-concept. A fully-fledged solution would implement the transport layer using sockets, IPC (Inter-Process Communication), or RPC (Remote Procedure Calls), depending on the requirements of the application.

Example 4.3. *Consider a simple Request-Reply protocol, where a participant sends a message to another participant, and receives another message in response.*

[4] An Abstract Base Class (ABC) in Python is a kind of class that is meant to be inherited from but is never meant to be instantiated. This is a design pattern that is heavily used in many large object-oriented programming languages.

```
1       class Transport(ABC):
2           async def send(self, part_id: str, msg: MessageType):
3               ...
4
5           async def recv(self, part_id: str) -> MessageType:
6               ...
7
8           async def peek(self, part_id: str) -> MessageType:
9               ...
10
11          def close(self):
12              ...
```

LISTING 3: Transport Abstract Base Class

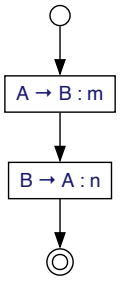


FIGURE 4.2: The Request-Reply protocol as a g-choreography

A g-choreography for the Request-Reply protocol is shown in Figure 4.2, and its projections are shown in Figure 4.3. To illustrate the expected message flow, we show in Figure 4.4 the LTS for said system.

Listing 4 shows a possible implementation for the Request-Reply protocol from Listing 1.

After each operation, components are updated to their respective continuation types (lines 2 and 7). For example, after the client sends a message, the client is updated to the type `Recv[Server, Text, Eps]`*, whereas the server is updated to* `Send[Server, Text,`

```
1       async def reqrep_client(client: ReqRepClient, msg: Text) -> None:
2           client = await client.send(msg)
3           client, reply = await client.recv()
4           return reply
5
6       async def reqrep_server(server: ReqRepServer) -> None:
7           req = await server.recv()
8           rep = process(msg)
9           await server.send(rep)
```

LISTING 4: Python Local Implementations of a Simple Request-Reply Protocol

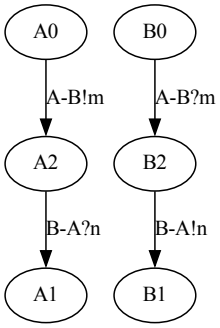


FIGURE 4.3: Local projections for the Request-Reply protocol

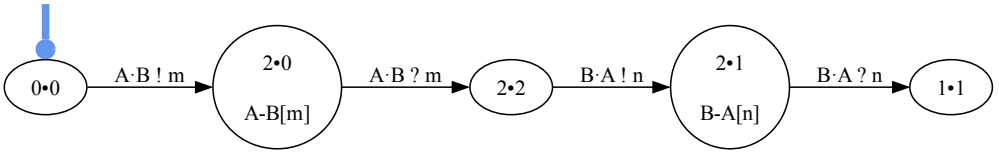


FIGURE 4.4: LTS for the Request-Reply protocol

`Eps`]. This allows a Python type checker (such as Pyright) to infer the correct continuation type returned. Thus, it is not necessary to explicitly annotate the type of the returned `LocalProtocol` variable for autocompletion to work or for the type-checks to be effective[5]*. This is particularly useful since very often the continuation type is complex and difficult to write out manually.*

An advantage of this approach is that it allows writing code that is very similar to the local type definitions, which makes it easier to understand the code and to verify that it is correct. The type checker offers immediate feedback on whether the correct sequence of send/receive operations is being performed, and whether the correct message types are being used at each step.

Example 4.4. *A possible implementation for the types from a protocol to check whether a number is even or not is given in Listing 5.*

The `offer` in line 4 type offers either a `Left` or `Right` instances which signal which branch of the protocol the sender chose. The main benefit of this approach is that type inference works on the parameters of the pattern matching expression (lines 6 and 8),

[5]Some type checkers like MyPy only allow for shadowing within the same block and nesting depth as the original definition [108], so this approach might not work for them.

```python
async def client(transport: Transport, n: int):
    t = EvenClient(transport)
    t = await t.send(number(n))
    offer = await t.offer()
    match offer:
        case Left(t, msg):
            t.close()
        case Right(t):
            t, msg = await t.recv()
            t.close()

async def server(transport: Transport):
    t = EvenServer(transport)
    t, msg = await t.recv()
    match msg:
        case number(n):
            if n % 2 == 0:
                t = await t.pick(no())
            else:
                t = await t.skip().send(yes())
    t.close()
```

Listing 5: Python Local Implementations of an "Is Even" Protocol

similarly to Rust enums. This, however, also has the drawback of making protocols with more than two choices slightly more cumbersome to represent, requiring nested binary trees for this purpose.

4.2.2 Property-based Conformance Testing

When testing whether a specific implementation conforms to a protocol specification, example-based testing is not necessarily suitable because the resulting trace from the execution of a communicating system is a sequence of send-receive actions, not a single value. Furthermore, the result of executing a protocol can depend on the order in which components execute their actions, making it difficult to test an implementation of a component in isolation. Instead, we want to test a protocol implementation by providing a set of properties that the protocol should always satisfy and generate random inputs to test the protocol. For this purpose, we can use the concepts defined in Chapter 3 to verify that the component implementations satisfy the specification, i.e., that the implementations are refinements of the specification.

Suppose we have a choreography specification, G, and a set of component implementations, $\{M_p\}_{p \in P}$, obtained from projecting the choreography onto each participant p. As mentioned in Chapter 3, if we want to test an implementation of a specific component, say M_i, we can take the remaining projections $\{M_p\}_{p \neq i}$, execute them in parallel with M_i, and check whether any of the properties derived from the specification are violated.

Hypothesis' strategies are utilized for two main purposes:

1. Generating random values for messages sent by test components (i.e., fuzzing).
2. Generating random choices when multiple actions are enabled across different participants.

Relying on Hypothesis for both purposes has multiple advantages. Firstly, it allows us to easily generate minimal failing examples: a failed test case will immediately retrace the exact sequence of choices and data that led the CUT to a faulty state. On the other hand, this also allows us the developer to decide which models to use as input to the test generator: either the split models from Chapter 3, or the non-deterministic models directly resulting from the projections, since Hypothesis will generate and register random choices for non-deterministic models at runtime.

Drawing inspiration from the `hypothesis.stateful` module, which provides a state machine abstraction for testing stateful systems [64], we can define a set of actions that can be performed on a system and generate random sequences of actions for testing. Instead of using Hypothesis's stateful module directly, we implement our custom loop since the `stateful` module is primarily designed for testing single state machines, while our tests are composed of multiple components interacting among themselves. Another reason for implementing a custom test loop is that the stateful module is designed to stop a test at arbitrary points, regardless of whether the system has reached a stable state or not.

For each local specification, we generate a test that checks the correctness of a specific component implementation. We define a set of state machines representing the global state of the system (except for the component being tested). At each step, we check the valid actions that each machine can perform. The validity of an action at a given point depends on whether the corresponding message type is ready to be received by a given machine, i.e., whether the message type is in the set of messages that the machine is waiting for (Send and Choice actions are always enabled). If an action is valid, we add it to a set of valid actions to be drawn from later. If no valid actions are found, we stop the test and check whether all test machines are in a stable state.

Both of these are *shrunk* as part of Hypothesis' testing mechanism [61]. Shrinking values means that if a generated value causes a test to fail, Hypothesis will try to find a minimal value that still causes the test to fail. Shrinking sequences of actions works similarly: if a sequence of actions leads to a state that violates a property, Hypothesis will try to find a minimal sequence of actions that leads to the same state and report this sequence as the failing test case. This makes it easier to reproduce the error and manually debug the

system when needed. Although the specific details of the shrinking process are out of the scope of this thesis, it is worth mentioning that each data type has its own individual shrinking strategy: these often gravitate towards the neutral element of each data type[6]. Recall that any failing test case is saved and prioritized in the subsequent test runs, to quickly verify whether the bug has been fixed.

4.2.3 Code Generation

We use a set of Jinja2[7] templates to generate both the types and the tests. The relevant components are split into different files to avoid circular dependencies in imports and to allow the developer to customize specific files, while allowing to regenerate the rest of the files directly from the protocol, keeping the code in sync with the model.

The code generation step receives as input a set of local specifications (i.e., a .fsa file) and outputs a set of files according to the following structure:

Implementations This file contains scaffolds for the implementations of each of the participants of the protocol. These empty implementations can be used as a starting point to implement the participants using the given local session types, or can be used as adapters to existing implementations. This file is not overwritten by default when regenerating tests, so the user can iterate on the implementation in case the tests fail and bugs are found.

Session Types This file contains the session types for each participant, as well as the initial types for each participant, which are aliases to the initial state of the corresponding machine in the Communicating Finite State Machines (CFSM) representation. This file is overwritten by default when regenerating tests, so the session types can stay in sync with the local specifications during development.

Types This file contains the message type classes and the participant type classes. This file is separate in order to avoid circular import dependencies. Initially these classes are empty, but they can be easily annotated with additional information: adding attributes with type annotations will impact the way Hypothesis generates random values while running the tests. These added attributes are kept by default.

Tests This file contains the entry points for the tests, or what could be referred in the literature as the "test harness" or the "test driver". This includes initializing the transports, instantiating the protocols with them, scheduling the execution of the participant implementation and executing the test loop with the rest of the

[6]For more information, the interested reader can refer to [61].
[7]Jinja2 is a popular templating engine for Python.

generated state machines. These tests are designed to be run using the `pytest` framework.

Test Machines This file contains the state machines that are used to test the implementations. This is further explained in the following section.

4.2.4 Test Machines

To generate a test machine for a given participant, its number (and type) of actions in each state is taken into account, according to the given model. For each state, either one or two methods are generated to account for the possible actions from that state. These methods are named according to the following convention:

`_<source_state>_<msg_type>_<action>_<target_state>`

For example, a method called `_2_n_send_1` will (given that we are in state 2) go to state 1 by sending a message of type `n`.

States without any actions are converted into methods which correspond to the `Eps` type, representing the end of a protocol. A single method is generated corresponding to this action, which simply closes the underlying transport. States with a single action are converted into methods which correspond to the `Send` or `Recv` types, depending on the type of action. A single method is generated, executing the corresponding action.

States with two or more actions are converted into methods which correspond to either the `Choice` type (if the first action is a "send") or an `Offer` type (if the first action was a "receive"). The remaining actions are recursively dealt with as above. In the case of an active choice, two methods are generated, one for each possible action: either the current action is executed, or it is skipped, and the protocol continues as the right continuation protocol expresses. The goal is for each of these to be dynamically selected at runtime by the test runner, so that the test runner can explore both branches of the choice.

Each of the generated methods first asserts that the current state of the protocol matches the expected state, and then:

- If the action is a "send", draw a random value for the message from the corresponding Hypothesis strategy, send the message using the protocol, and update the state to the target state.

- If the action is a “receive”, receive a message using the protocol, assert that the received message is of the expected type, and update the state to the target state.
- If the action is an “offer”, receive the message, assert that the received message is of the expected type, and update the state to the target state.
- If the action is a “choice”, receive a choice using the protocol, assert that the received choice is of the expected type, and update the state to the target state.
- If the action corresponds to an `Eps` state, close the underlying transport.

The generated asynchronous methods are used in the test machine classes to handle different protocol actions based on the current state and type expression.

4.2.5 The Test Driver

The preceding sections have resulted in a final series of tests, which can be executed using the `pytest` framework. Each test method, as devised, ensures the correct models are loaded for the respective abstract tests and integrates them with an asynchronous test loop. This loop is a key component of what is commonly termed as the 'test driver' or 'test harness' in software testing literature [112]. It handles CFSMs, and uses the test machines to simulate participants in the given protocol, placing particular emphasis on the CUT.

During the test loop, the method iterates through the *active transitions*, selecting one randomly to be executed. These transitions are responsible for driving the interactions between the different participants in the protocol. Each executed transition triggers the corresponding functionality within the associated test machine. The method ensures that only the appropriate actions defined by the session types are performed, validating the correctness of the interactions.

As the test loop progresses, the method checks for specific conditions. The specific properties that are tested throughout this loop are:

1. The system does not deadlock.
2. The system eventually reaches an accepting state.
3. Optionally (depending on whether the black-box assumption is enacted or not) the framework also checks if the CUT raises an exception.

If an accepting stable state is reached, the test loop terminates.

4.3 Evaluation

To test the validity of our approach, we have extended ChorTest to allow for the generation of test cases from protocols given as g-choreographies. Together with the process from Chapter 3, the whole workflow takes as input a choreography specification, outputting a set of test cases for specific components implementing the choreography. Table 4.1 shows the relation between the abstract concepts from Section 3.5 and their counterparts in our reference implementation. These functionalities are available as part of ChorTest as additional subcommands from the CLI interface, and are also supported by using a VSCode extension, which we briefly introduce in Section 4.4.

Framework concept	**Reference**	**Alternatives**
Abstract projection	As drawn from [48]	[24, 113]
Abstract well-formedness	As drawn from [48]	[25, 51]
Test case	Each pytest test case represents a set of determinized abstract test cases. Splitting occurs implicitly.	e.g., encode test cases as mappings close to the original CFSMs
Oracle scheme	ChorTest default : accepting states are those with no outgoing transitions in test CFSMs	Manually defined ones, or more complex auto-generated ones

TABLE 4.1: Mapping of abstract framework concepts to their counterparts in the reference implementation

We examine our test generation process through a series of examples. We also show how it can be used to test the validity of a protocol implementation. Finally, we evaluate the effectiveness of our tool according to the following metrics:

Line and branch coverage We measure the *line* and *branch* coverage of the whole test suite on the implementation code. In our context, line coverage refers to the lines of code of the implementation that are executed by the tests, while branch coverage refers to the total number of branches that are executed[8]. Although full coverage is no guarantee of correctness, higher coverage is generally preferable, as it means that more of the implementation code is being tested.

Message, state, and transition coverage We also report the *message*, *state*, and *transition* coverage of the test suite. These metrics refer to the ratio of messages

[8]For instance, a conditional statement with two branches will have a branch coverage of 50% if only one of the branches is executed.

exchanged out of the possible types, as well as which states and transitions visited by the test machines, out of all the possible ones.

Fault detection We evaluate our ability to identify faults or errors in the implementation code. We inject different types of errors and measure whether we can detect each of them.

Performance We consider the time to execute the tests regarding the performance of the tests, such as the time to failure for those failures detected.

To assess the fault detection capabilities of ChorTest , we manually inject faults into the implementation code, and measure the ability of the generated tests to detect them. We consider the following types of faults:

1. The CUT sends a message of the wrong type.
2. The CUT sends the correct message, but with an incorrect payload.
3. The CUT is unresponsive/deadlocked.

Notice that these fault types (except for the one changing the message type) do not have an equivalent to the ones at the abstract level from Section 3.5. Instead, they are specific to the implementation code, and are only detectable at the concrete level.

Example 1: The Request-Reply Protocol

To illustrate a full end-to-end usage of the framework, we recall the Request-Reply protocol introduced previously, where two participants exchange messages of type *Text*.

The generated test machine for the server participant is shown in Listings 6[9].

In Listing 6, we can see that each transition is named after the message that triggers it, and that the payload of the message is generated using the Hypothesis library (more specifically, the `builds` strategy). This strategy can automatically infer the correct data type for the message payload, based on the type hints provided in the "Types" file mentioned before. This given type can be a primitive type, such as `int` or `str`, or a custom type, such as the `Text` type defined in Listing 4. By default, these are generated as empty types, but they can be easily extended to include more complex types.

[9]Note that the generated test machine for the client participant is simply the binary dual to the one shown here. For the sake of simplicity, we omit the assertions checking that the protocol type is the correct one, and the assertions checking that the state is the expected one at each point.

```
1
2           class ServerTestMachine(TestMachine):
3               b: LocalProtocol
4               data: st.DataObject
5
6               def __init__(self, b: LocalProtocol, data: st.DataObject):
7                   ...
8
9               async def _0_m_recv_2(self):
10                  self.protocol, _msg = await self.protocol.recv()
11                  self.state = 2
12
13              async def _2_n_send_1(self):
14                  _msg = self.data.draw(st.builds(n))
15                  await self.protocol.send(_msg)
16                  self.state = 1
17
18              async def _1_eps(self):
19                  self.protocol.close()
20
```

Listing 6: Generated Test Machine for the Server Participant for the Request-Reply Protocol

```
1           async def b(transport: Transport):
2               t = BType(transport)
3               t, _msg = await t.recv()
4               raise Exception("Injected failure")
5               t = await t.send(n())
6               t.close()
```

Listing 7: ReqRepServer with an Injected Failure

The `data` object is an instance of the `DataObject` class, which allows to interactively draw from different strategies. It is the same object used to draw actions from the available ones. An advantage of this is that, when an example fails, the `data` object is automatically shrunk to the smallest possible example that still reproduces the failure, shrinking both the sequence of choices and the payload of the messages.

We implemented a simple `AsyncioTransport` using the `asyncio` library, which runs the tests in a single thread while controlling the non-determinism arising from the scheduler. This has the upside of allowing us to reliably reproduce a specific error scenario caused by a potential data race from the test machines' side, for example.

Recall Listing 4 for the implementation of the Request-Reply protocol in Python. We inject some failures by raising an exception at each step of the implementation code (such as the one shown in Listing 7 — line 4). We then run the generated test machines against the faulty implementation, and measure the ability of the tests to detect the injected failures.

Executing the tests shows that (as discussed in Chapter 3) the generated tests can detect failures which occur before the last message is sent, but not after. Intuitively, this is because the last message sent from the CUT will alter the state of at least one test machine, and any test machine will not be able to detect any further failures after this last message was sent. This is not a limitation of the framework, but rather a limitation of testing under a black-box setting.

Now, let us address the scenario where the Component Under Test (CUT) inadvertently transmits a message of an incorrect type. Initial protection against this is offered by our multiparty session library, which imposes static checking on message types and thereby prevents the CUT from transmitting messages of an incorrect type. However, for the sake of thoroughness, let us hypothesize a situation where the CUT manages to transmit an incorrectly typed message — perhaps the CUT is a third-party service not developed within our Multiparty Session Types (MPST) framework. In such an event, our testing architecture is fully equipped to identify these discrepancies: as the test machines anticipate a message of a certain type, a mismatch with the actual message type sent by the CUT will signal a failure in the test.

Identifying errors where the CUT dispatches a flawed payload (but with the correct message type) needs assertions on the data obtained by the test machines. Currently, this can be achieved; however, it requires manual modifications to the auto-generated test machines. While the potential for automatically generating these assertions is not dismissed, the actual implementation of this feature is a subject for future development.

Failures where the CUT is unresponsive/deadlocked are detected thanks to a timeout mechanism, which is automatically taken care of during the test execution. These timeouts can be configured by the user, and are set to 0.01 second by default in our experiments, which proved enough to yield to the test machine coroutines. From the 2 tests required, where a successful test normally takes around 5ms each, once failures are injected the runtime grows to around 30ms, mostly due to the timeouts on the *receive* actions of the test machines.

Since this example is just for illustrative purposes, we do not report detailed metrics, since the tests achieve full coverage after a single run of the test suite for each CUT.

Example 2: The ATM Protocol

For our next example, let us recall the full ATM protocol introduced in Figure 3.3. From the .fsa specification, we can generate executable code in two different ways: either through the command in the VSCode extension, or by using the CLI tool. This will

```python
async def b(transport: Transport):
    t = BType(transport)
    t, msg = await t.recv()
    t = await t.skip().send(granted())
    off = await t.offer()
    match off:
        case Left(t, msg):
            t = await t.pick(allow())
            t.close()
        case Right(t):
            match await t.offer():
                case Left(t, msg):
                    t = await t.send(balance(10))
                    t.close()
                case Right(t):
                    t, msg = await t.recv()
                    t.close()
```

Listing 8: An Example Bank Implementation to be Tested

generate a folder following the structure mentioned previously in Section 4.2.3, with the generated code for each participant in the protocol.

The generated code includes templates for implementing individual components representing each participant in the protocol. These templates serve as a starting point for developers, who can easily transform them into complete implementations by following the actions defined in the auto-generated session types. By employing a static type checker like Pyright, the system ensures that only the appropriate actions are executed at each step of the protocol, emitting a type-check warning if an incorrect action is performed.

This approach not only guarantees the execution of the correct actions but also streamlines the protocol implementation process. With the aid of autocompletion, developers can efficiently navigate and complete the implementation stage, leveraging the previously designed protocol.

To assess the usefulness of the generated code in finding bugs that do not break compliance, we will go through a possible implementation of the bank component and analyse the results from running the generated tests against it. Testing the implementation from Listing 8 results in no errors and all tests pass. However, even though there is no error raised by the type checker, a careful observer might notice a few missing transitions, specifically A C!*bye* and B A?*deny*. What other information could we use to detect these missing transitions?

Although we've considered the generated tests to be black-box tests, we can still use the coverage metrics to get an intuition of where the bug might be. The coverage metrics at the model level are reported in Figure 4.5. The graph reflects the increase in each

respective type of coverage as the tests progress. For this example, we set Hypothesis' *max_examples* parameter to 100, meaning that each test will run until 100 examples have been generated. As a result, since for this case there are three participants and one generated test methods each, the total number of tests run is 300. The bumps in coverage every 100 tests are expected, due to the fact that the tests are executed sequentially. Once the first test is done, the second one starts from scratch, covering transitions in test machines that were not covered by the first test.

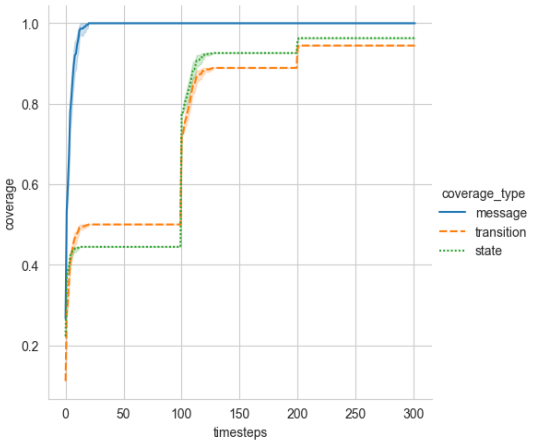


FIGURE 4.5: Initial ATM Protocol coverage

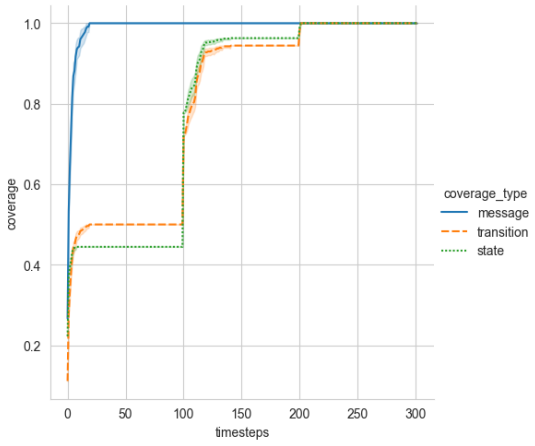


FIGURE 4.6: Corrected ATM Protocol coverage

Metric	Faulty Bank	Corrected Bank
Covered Lines	65	69
Num Statements	66	69
Line Coverage	0.984	1
Num Branches	20	22
Covered Branches	18	20
Branch Coverage	0.900	0.909

FIGURE 4.7: Code Coverage for the ATM Example

These transitions can be traced back in the g-choreography to the branch where the bank has an internal choice, where it should send either `allow` or `deny` messages. This might be indicating of there being a bug in the implementation of the bank, and more specifically at the point where the request is authenticated and the bank decides whether to allow or deny it. Analysing missing transitions in coverage and tracing their way back in the choreography gives us an intuition of where the bug might be.

By simply tracing the execution back to the first internal choice where the active participant might not be picking a branch, we've found the bug. After correcting the bank implementation to make this decision properly, as reflected in Listing 10 (lines 5 and 8-10), the metrics shown in Figure 4.6 change to reflect that these uncovered transitions

```
async def b(transport: Transport):
    # [...]
    match off:
        case Left(t, msg):
            t = await t.pick(allow())
            t.close()
    # [...]
```

Listing 9: Incorrect Bank Implementation Fragment

```
async def b(transport: Transport):
    # [...]
    match off:
        case Left(t, msg):
            if authenticate(msg):
                t = await t.pick(allow())
                t.close()
            else:
                t = await t.skip().send(deny())
                t.close()
    # [...]
```

Listing 10: Corrected Bank Implementation Fragment

are now being executed by some test. The increase in coverage reflects the bug has been fixed.

The coverage metrics at the code level for this example are reported in Table 4.7. Although this information is not always readily available (as is often the case with black-box testing), it is surely useful to have a more detailed view of the coverage achieved by the tests. In this case, we can see that the faulty implementation is missing a line. Inspecting the code and the coverage reports offered by the CLI, it can be seen that the body of the `authenticate` method is not covered by the tests. Note that we were able to arrive at this same conclusion by simply looking at the coverage metrics at the model level, which shows that our approach is able to provide useful information even when the code is not available.

We report the execution times over 30 runs in Figure 4.8.

4.4 Discussion

The generation of executable tests from local session types effectively translates the abstract local protocols introduced in previous chapters into actionable test sequences, providing an automated and reliable means of ensuring system components' adherence to the original communication protocol.

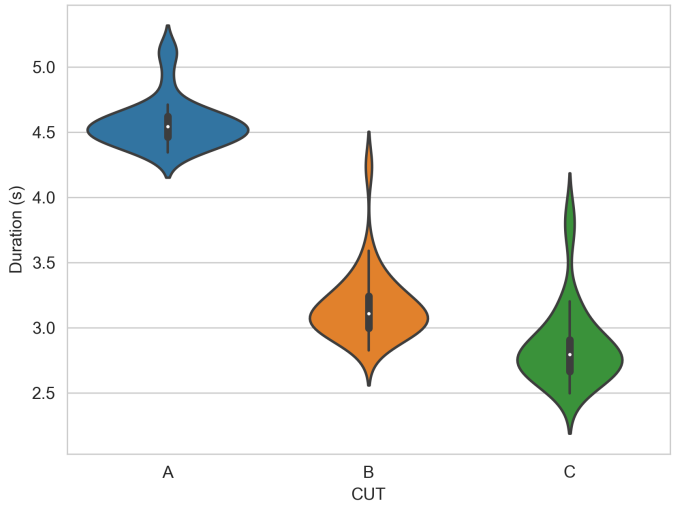


FIGURE 4.8: Execution Times for the ATM Protocol tests for each Participant

Compared to existing tools and methodologies, our approach introduces the concept of using local session types as a foundation for generating tests. This offers a novel perspective that could streamline the testing process while enhancing its effectiveness. For instance, in contexts where communication protocols are highly dynamic or complex, our current model may require additional refinement to accurately generate test sequences.

By automating the generation of tests, ChorTest not only reduces the workload on software developers but also potentially increases the reliability of tests by minimizing human error in test creation. That said, there are practical considerations to bear in mind. The generation of data for the test cases is currently random, which may not be ideal for all scenarios. For instance, in cases where the user would like to restrict the range of values for a given parameter (e.g., a specific range of integers), our tool would need to be extended to accommodate for custom Hypothesis strategies.

In addition to the contributions outlined in this chapter, practical tools have been developed to assist with the proposed workflow. One such tool is a Visual Studio Code extension specifically designed to facilitate the editing and visualization of .gc and .fsa files. This extension provides basic syntax highlighting, enabling a more readable and manageable editing environment. It also includes minimal snippets for easier editing, which enhances the user experience and speeds up the development process. The extension further enriches the workflow by providing an automatic preview of .gc and .fsa files upon save, which can significantly aid in maintaining a clear understanding of the system's current state. Moreover, it offers command options for projection, LTS generation, and code generation, effectively integrating the proposed workflow into a widely used development environment. The extension is openly accessible and can be found on

GitHub and the VSCode Marketplace. This tool embodies the proposed workflow, and further illustrates the feasibility and applicability of our approach.

Another highlight in terms of accessibility and ease of use of the tool is the inclusion of a Dockerfile in the project's repository, as well as devcontainer configuration files, which can bring several substantial benefits. Firstly, it streamlines the setup process: Docker encapsulates the tool's dependencies within a container, ensuring that the tool runs reliably across different environments. This pushes towards mitigating the "works on my machine" problem, saving time and reducing frustration for developers who want to use or contribute to the tool. Secondly, the devcontainer configuration allows for an even more seamless development experience when used with tools like Visual Studio Code. It automatically sets up a development environment with the correct dependencies, tools, and configurations in an isolated container. This means that new developers or contributors can get started quickly without needing to worry about setting up their environment correctly. Furthermore, it helps maintain consistency in development environments across the team, reducing the potential for environment-specific bugs and disparities.

Regarding the scalability of our approach, while our tool is designed to support a wide range of protocols, it is not yet optimized for large-scale systems. In particular, g-choreographies with a high degree of parallel interleaving will result in large communicating finite state machines, which can be difficult to handle systematically. At the abstract level, this can become an issue particularly when generating, for instance, the labelled transition systems from large local projections: it might turn out that the actual LTS is too large to be systematically generated. However, even if this were not feasible, our abstract test generation process can still be used to generate a subset of the possible tests. Furthermore, in our reference implementation, we account for this by not laying out explicitly all possible interleavings of the test machines, but by exploring them dynamically and stochastically as the tests are executed. By using coverage as a metric to guide the exploration of the internal-choice decision space and maximizing the number of transitions explored, we ensure that the generated tests are as effective as possible at finding bugs. This allows us to generate tests for protocols that would otherwise be too large to be practical.

One significant constraint in our study is the need for a well-branched and well-formed g-choreography as a prerequisite for test generation. Should a system's choreography not meet these conditions, our tool's utility would be limited. Furthermore, although our tool is designed to support a wide range of programming languages, our current multiparty session library prototype is implemented in Python. It would take additional

effort to extend our tool to support other languages, although the models themselves can be reutilized for this purpose.

In terms of future work, these limitations present potential avenues for exploration. For instance, refining our tool to handle a wider range of choreographies, even those that may not strictly meet the well-branchedness or well-formedness conditions, would enhance its utility. Additionally, extending our tool to support other programming languages would broaden its applicability. We also believe it is possible to generate the tests without explicitly generating methods, but rather by generating a single test method that would be able to dynamically execute all the transitions in the protocol. This could potentially reduce the amount of code generated, and make the tests more readable, while still allowing for more flexibility by allowing the user to override a specific transition or a set of transitions.

Chapter 5

Discussion

We presented an abstract framework to support model-driven testing of message-passing systems based on choreographies. The framework relies on the so-called top-down approach featured by an existing choreographic model. More precisely, we exploited the notion of projection of global views of choreographies in order to devise an automatic test generation mechanism. The design of our algorithm required us to fix the basic notion of test, test compliance, and test success within the framework of g-choreographies and communicating systems. We also extended the theoretical framework with a prototype implementation, and a mutation testing-based evaluation of the generated test cases. The results of the evaluation show that the generated test cases are able to detect faults in the implementation of the system under test. Later on, we also proposed a set of Python libraries to support the development of distributed systems based on choreographies. The libraries are based on the notion of local session types, providing a simple and intuitive interface to the developer. From the local models derived from our theoretical framework, we then proceeded to generate executable Python tests that can be used to test the implementation of specific components of the system.

Some optimizations to reduce the number of tests are possible. A first optimization can be the reduction of internal choices generated by the parallel composition. In fact, those tests are redundant and one would be enough in the semantics of communicating systems adopted here (where channels are multisets of messages). It is important to observe though that the redundancy of tests depends on the chosen semantics of communicating systems. For instance, the optimization above does not apply when the communicating systems interact through FIFO queues.

Instead of concretizing abstract tests, one could think of extracting CFSMs from actual implementations and running the tests on them. Machines could potentially be extracted directly from source code. If, however, the source code was not available, it

could still be possible to test components (e.g., by employing techniques from the field of process mining to infer the CFSMs from available data, such as traces [114]). Note that such techniques should be more efficient than concretization (because it does not let abstract tests proliferate into many concrete ones). Moreover, another advantage of this approach could be that it enables us to exploit the bottom-up approach of choreographies, where global views are synthesized from local ones [113]. The synthesized choreography can be compared with a reference one to derive tests that are more specific to the implementation at hand.

We have made multiple design choices when devising our framework, but alternative approaches are possible. Firstly, different notions of test cases are possible that would induce the adoption of different algorithms for test generation. For instance, a natural alternative is to take the projection of one component as the CUT, say M, and consider as test cases the CFSM obtained by *dualizing* M, that is by taking a CFSM isomorphic to M where each output is replaced by the corresponding input and vice versa. Note that this yields a non-local CFSM as a test case; we preferred to explore first an approach which yields "standard" communicating systems.

Definition 3.7 formalizes when a test case is meaningful for a choreography. It would be also desirable to relate traces of test-compliant machines with the language of the choreography. Ideally, for a choreography G an *adherent* test T should guarantee that for every T-*compliant* machine M the traces of runs of $M \otimes T$ that end in a successful configuration are in $\mathcal{L}[\mathsf{G}]$. This property cannot be guaranteed by our framework for arbitrary choreographies. Firstly, the CUT may force causal relations. For example, consider $\mathsf{G} = \mathsf{A}{\to}\mathsf{B}\colon \mathtt{x}; \mathsf{B}{\to}\mathsf{A}\colon \mathtt{y}; \mathsf{A}{\to}\mathsf{C}\colon \mathtt{z}$, where the CUT is a realization of A. According to G, the event $\mathsf{B}\,\mathsf{A}!y$ should always precede $\mathsf{A}\,\mathsf{C}!z$. However, this dependency is enforced by A. For instance, let M be a machine for A that sends these events in the opposite order, no test obtained from the machines for B and C would be able to detect the wrong execution of M unless some explicit communication between B and C is added. Secondly, in an asynchronous setting it may be impossible to distinguish some behaviours of the CUT. For example, in $\mathsf{A}{\to}\mathsf{B}\colon \mathtt{x}; \mathsf{A}{\to}\mathsf{B}\colon \mathtt{y}$ the event $\mathsf{A}\,\mathsf{B}!x$ should always precede $\mathsf{A}\,\mathsf{B}!y$, but this order is not observable by B in case of asynchronous communication (unless FIFO buffers are used).

In summary, the notion of adherence is not enforceable for all g-choreographies or all possible realizations of the participant to be tested. This hints to the following open problems:

- the identification of a proper notion of adherence in an asynchronous setting,

- the identification of "interesting" subclasses of g-choreographies for which the strict notion of adherence is meaningful, and
- the extension of the testing framework to enforce such notion, either by adding communications between components or by using non-local machines.

The operational semantics used in ioco-testing [80] permits inspecting when the CUT cannot perform some actions. This is not required in our context: tests are generated purely from global specifications without making assumptions on the behaviour of the CUT. The use of a global specification could also allow for other interesting applications, such as studying at the global level how a specific non-compliant component violates the global protocol, which might help correct common mistakes in implementations.

At a first glance, our approach corresponds to the must-preorder of the testing theory of De Nicola and Hennessy [82]. In fact, the notion of test compliance (cf. Definition 3.6) imposes conditions on all the maximal runs of the CUT in parallel with the test. However, there are two key differences between the theory of testing and our approach which make a precise analysis non-trivial. The first difference is that we consider asynchronous communications and the second is that our tests are "multiparty", namely tests are obtained by composing many CFSMs. It might be that the results of Boreale et al. [115], which extend to asynchronous communications the classical theory of testing, can be combined with the work of De Nicola and Melgratti [116] to give a suitable theoretical setting to our framework.

Regarding coverage, it is worth noting that multiple notions have been defined in the literature of software testing. In general terms, coverage refers to how much a specific criterion is satisfied by a set of test cases. For instance, *statement coverage*, *branch coverage*, *path coverage*, and *mutation adequacy* have been used to measure the quality and adequacy of test suites [117]. Within specification-based testing, Offutt et al. [118] mention several criteria that can be evaluated on test suites based on finite state machines: *transition coverage*, *full predicate coverage*, *transition-pair coverage* and *complete sequence*. We believe our approach targets the complete sequence criterion as they define it, in the sense that each test case generates "meaningful sequences" that test engineers can choose (based on their experience and domain knowledge) to check the system.

Since message-passing systems fall under the class of *reactive systems*, we drew inspiration from the work done on model-driven testing of reactive systems [55]. In particular, we showed that choreographies can, at least to some extent, be used to automatically generate executable tests and as test case specifications [119]. Technically, we exploited

the so-called *projection* operation of choreographic models. Another concrete projection was formalized for the first time in [24] (for multiparty session types) and for g-choreographies in [26, 27, 48], elaborating on the projection of global graphs [120].

In this work, we consider component testing. The level of granularity we adopt implies that participants are components, and our framework is designed to test a single component at a time. An intriguing open problem is to apply our framework to support integration testing [121]. In fact, one could think of defining *group* projections, namely projection operations that generate communicating systems representing the composition of several participants. We believe that this approach could pay off when the group onto which the g-choreography is projected can be partitioned in a set of "shy" participants that interact only with participants within the group and others that also interact outside the group. The former set of participants basically corresponds to units that are stable parts of the system that and do not need to be (re-)tested as long as the components in the other group pass some tests.

Deriving tests from a formal model allows us to ask ourselves if, once suitability is established, it is possible to ascertain some properties of the system under test, such as compatibility. Several notions of compatibility for CFSMs have appeared in the literature (e.g. [113, 122–126]). One could wonder if the intuitive compatibility notion stating that *a CFSM which passes all tests is compatible with the rest of the system* could be reasonable. In other words, whether our tests can guarantee a similar (albeit more restricted) form of compatibility. Since our tests cannot detect all possible misbehaviours (as shown in Chapter 3), and, in general, complete testing is practically unfeasible once data is considered, we believe that it might not be possible to directly prove properties like the aforementioned ones. However, it does seem interesting to study whether it is possible to obtain some guarantees from a complete abstract test suite, in the shape of a (possibly weaker) notion of compatibility.

On a similar line, it is worth noting that compliance and suitability do not imply language inclusion, since, as hinted before, passing tests does not necessarily guarantee the absence of bugs. An intriguing question is whether the opposite holds: does test non-compliance imply language non-inclusion? Our abstract framework is too liberal to address this question since it does not require specific relations between the language of the choreography and the test cases. Thus, we believe that the answer would be no, in general. If, however, we restrict ourselves to our specific test generation approach, we conjecture that the suggested implication may actually hold when considering maximal traces. We plan to resolve these final questions as future work.

Bibliography

[1] Phillip Johnston and Rozi Harris. The boeing 737 max saga: lessons for software organizations. *Software Quality Professional*, 21(3):4–12, 2019.

[2] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. doi: 10.1109/MC.1993.274940.

[3] Nancy G. Leveson. The therac-25: 30 years later. *Computer*, 50(11):8–11, 2017. doi: 10.1109/MC.2017.4041349.

[4] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007. ISBN 978-0-471-45593-6.

[5] Boris Beizer. *Software Testing Techniques*. Intern. Thomson Computer Press, London, 2. ed edition, 1990. ISBN 978-1-85032-880-3.

[6] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, 2008. ISSN 00010782. doi: 10.1145/1378727.1378742.

[7] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, 2009. doi: 10.1145/1592434.1592436.

[8] David A. Carrington and Phil Stocks. A tale of two paradigms: Formal methods and software testing. In Jonathan P. Bowen and J. Anthony Hall, editors, *Z User Workshop, Cambridge, UK, 29-30 June 1994, Proceedings*, Workshops in Computing, pages 51–68. Springer/BCS, 1994. doi: 10.1007/978-1-4471-3452-7_4.

[9] Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Trans. Software Eng.*, 44(8):747–783, 2018. doi: 10.1109/TSE.2017.2707089.

[10] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, Amsterdam ; Boston, 2007. ISBN 978-0-12-372501-1.

[11] Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. `https://www.w3.org/TR/ws-cdl-10/`. W3C Candidate Recommendation 9 November 2005.

[12] Object Management Group. Business Process Model and Notation. `http://www.bpmn.org`.

[13] Jonas Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018. ISBN 9781491994351.

[14] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. Cyber-physical microservices: An iot-based framework for manufacturing systems. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 232–239, 2018.

[15] Leonardo Frittelli, Facundo Maldonado, Hernán C. Melgratti, and Emilio Tuosto. A choreography-driven approach to apis: The opendxl case study. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2020. doi: 10.1007/978-3-030-50029-0_7.

[16] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, and Massimo Tivoli. Chorevolution: Automating the realization of highly-collaborative distributed applications. In *COORDINATION*, pages 92–108. Springer, 2019. doi: 10.1007/978-3-030-22397-7_6.

[17] Laura Bocchi, Hernán C. Melgratti, and Emilio Tuosto. On resolving non-determinism in choreographies. *Log. Methods Comput. Sci.*, 16(3), 2020. doi: https://doi.org/10.23638/LMCS-16(3:18)2020.

[18] Matthias Güdemann, Pascal Poizat, Gwen Salaün, and Lina Ye. Verchor: A framework for the design and verification of choreographies. *IEEE Trans. Serv. Comput.*, 9(4):647–660, 2016. doi: 10.1109/TSC.2015.2413401.

[19] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *International Conference on Software Engineering and Formal Methods*, pages 323–332. IEEE Computer Society, 2008. doi: 10.1109/SEFM.2008.11.

[20] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 191–202. ACM, 2012. doi: 10.1145/2103656.2103680.

[21] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi: 10.1145/2429069.2429101.

[22] Marco Autili, Paola Inverardi, and Massimo Tivoli. Automated synthesis of service choreographies. *IEEE Softw.*, 32(1):50–57, 2015. doi: 10.1109/MS.2014.131.

[23] Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In Christel Baier and Luís Caires, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10854 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2018. doi: 10.1007/978-3-319-92612-4_2.

[24] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *JACM*, 63(1):9:1–9:67, 2016. ISSN 0004-5411. doi: 10.1145/2827695. Extended version of a paper presented at POPL08.

[25] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 26(2):238–302, 2016. doi: 10.1017/S0960129514000188.

[26] Roberto Guanciale and Emilio Tuosto. An Abstract Semantics of the Global View of Choreographies. *Electronic Proceedings in Theoretical Computer Science*, 223: 67–82, sep 2016. ISSN 2075-2180. doi: 10.4204/EPTCS.223.5.

[27] Roberto Guanciale and Emilio Tuosto. Semantics of global views of choreographies. *Journal of Logic and Algebraic Methods in Programming*, 95:17–40, 2018. doi: 10.1016/j.jlamp.2017.11.002.

[28] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Mauro Jacopo. Dynamic choreographies - safe runtime updates of distributed applications. In *COORDINATION 2015*, pages 67–82, 2015. doi: 10.1007/978-3-319-19282-6_5.

[29] Mario Bravetti and Gianluigi Zavattaro. Contract Compliance and Choreography Conformance in the Presence of Message Queues. In Roberto Bruni and Karsten Wolf, editors, *Web Services and Formal Methods*, volume 5387, pages 37–54. Springer Berlin Heidelberg, Berlin, Heidelberg, oct 2009. ISBN 978-3-642-01363-8 978-3-642-01364-5. `http://link.springer.com/10.1007/978-3-642-01364-5_3`.

[30] Alex Coto, Roberto Guanciale, and Emilio Tuosto. Choreographic development of message-passing applications - A tutorial. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2020. doi: 10.1007/978-3-030-50029-0_2.

[31] Alex Coto, Roberto Guanciale, and Emilio Tuosto. On testing message-passing components. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2020. doi: 10.1007/978-3-030-61362-4_2.

[32] Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.*, 123:100712, 2021. doi: 10.1016/j.jlamp.2021.100712.

[33] Wenjie Xu, Lin Chen, Chenghao Su, Yimeng Guo, Yanhui Li, Yuming Zhou, and Baowen Xu. How well static type checkers work with gradual typing? a case study on python. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 242–253. IEEE, 2023.

[34] Alex Coto, Franco Barbanera, Ivan Lanese, Davide Rossi, and Emilio Tuosto. On formal choreographic modelling: A case study in EU business processes. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, volume 13701 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2022. doi: 10.1007/978-3-031-19849-6_13.

[35] Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[36] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-26278-4. doi: 10.1007/b137241.

[37] Simon Gay and António Ravara. *Behavioural Types: from Theory to Tools*. River Publishers, sep 2017. ISBN 9788793519817. doi: 10.13052/rp-9788793519817.

[38] Hans Hüttel, Emilio Tuosto, Hugo Torres Vieira, Gianluigi Zavattaro, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, and António Ravara. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):1–36, oct 2016. ISSN 03600300. doi: 10.1145/2873052.

[39] Rumyana Neykova. Session Types Go Dynamic or How to Verify Your Python Conversations. *Electronic Proceedings in Theoretical Computer Science*, 137:95–102, oct 2013. ISSN 2075-2180. doi: 10.4204/EPTCS.137.8.

[40] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, oct 2015. ISSN 0925-9856, 1572-8102. doi: 10.1007/s10703-014-0218-8.

[41] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. *WGP 2015 - Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, co-located with ICFP 2015*, pages 13–22, 2015. doi: 10.1145/2808098.2808100.

[42] Simon Fowler. Session types in programming languages - a collection of implementations, 2016. `http://simonjf.com/2016/05/28/session-type-implementations.html`. Accessed: 2019-11-19.

[43] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi: 10.1145/2827695. A preliminary version of this article appeared in Proceedings of 35th annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2008).

[44] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8358 LNCS: 22–41, 2014. ISSN 16113349. doi: 10.1007/978-3-319-05119-2_3.

[45] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[46] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211, pages 194–213. Springer Berlin Heidelberg, Berlin, Heidelberg, sep 2012. ISBN 978-3-642-28868-5 978-3-642-28869-2.

[47] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *ACM SIGPLAN Notices*, volume 50, pages 221–232. ACM, 2015.

[48] Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *Journal of Logical and Algebraic Methods in Programming*, 95:17–40, sep 2018. ISSN 23522208. doi: 10.1016/j.jlamp.2017.11.002.

[49] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM (JACM)*, 1983. ISSN 1557735X. doi: 10.1145/322374.322380.

[50] Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, November 2005. ISSN 08905401. doi: 10.1016/j.ic.2005.05.006.

[51] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 435–446. ACM, 2011. doi: 10.1145/1926385.1926435.

[52] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Pearson Addison Wesley, 3rd edition, 2007. ISBN 0321455371, 9780321455376.

[53] Pierre Bourque, R. E Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge.* 2014. ISBN 978-0-7695-5166-1.

[54] Jan Tretmans. Model-based testing: Property checking for real. In *International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices*, 2004.

[55] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *LNCS*, 2005. Springer. doi: 10.1007/b137241.

[56] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *5th International Conference on Engineering of Complex Computer Systems (ICECCS '99), October 18-22, 1999, Las Vegas, NV, USA*, page 119. IEEE Computer Society, 1999. doi: 10.1109/ICECCS.1999.802856.

[57] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction, October 1991.

[58] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, 2008. ISBN 978-0-521-88038-1.

[59] Larry J. Morell. *A Theory of Error-Based Testing.* Technical report, University of Maryland, College Park, MD, 1984.

[60] Jeff Offutt. *Automatic Test Data Generation.* Technical report, Georgia Institute of Technology, Atlanta, GA, 1988.

[61] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 13:1–13:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.13.

[62] David Maciver. The hypothesis documentation. Online: `https://hypothesis.readthedocs.io/en/latest/` (Accessed on Feb 2023), 2013.

[63] David Maciver and Zac Hatfield-Dodds. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*, 4(43):1891, 2019. doi: 10.21105/joss.01891.

[64] David Maciver. The hypothesis documentation - stateful module. Online: `https://hypothesis.readthedocs.io/en/latest/stateful.html` (Accessed on Feb 2023), 2013.

[65] Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019. ISSN 2475-1421. doi: 10.1145/3290343.

[66] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 56–71. Springer, 2012.

[67] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[68] Alain Finkel and Etienne Lozes. Synchronizability of communicating finite state machines is not decidable. *arXiv preprint arXiv:1702.07213*, 2017.

[69] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 273–284, 2008. ISSN 07308566. doi: 10.1145/1328438.1328472.

[70] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. A tool for choreography-based analysis of message-passing software. *Behavioural Types: from Theory to Tools*, page 125, 2017.

[71] BPMN Specification - Business Process Model and Notation. https://www.bpmn.org/.

[72] Flavio Corradini, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Collaboration vs. Choreography Conformance in BPMN 2.0: From Theory to Practice. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 95–104, Stockholm, October 2018. IEEE. ISBN 978-1-5386-4139-2. doi: 10.1109/EDOC.2018.00022.

[73] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Global vs. Local Semantics of BPMN 2.0 OR-Join. page 15.

[74] Flavio Corradini, Alessio Ferrari, Fabrizio Fornari, Stefania Gnesi, Andrea Polini, Barbara Re, and Giorgio O. Spagnolo. A Guidelines framework for understandable BPMN models. *Data & Knowledge Engineering*, 113:129–154, January 2018. ISSN 0169023X. doi: 10.1016/j.datak.2017.11.003.

[75] David Raymond Christiansen, Marco Carbone, and Thomas Hildebrandt. Formal semantics and implementation of bpmn 2.0 inclusive gateways. In *International Workshop on Web Services and Formal Methods*, pages 146–160. Springer, 2010. doi: 10.1007/978-3-642-19589-1_10.

[76] Mario Cortes-Cornax, Sophie Dupuy-Chessa, and Dominique Rieu. Choreographies in bpmn 2.0: new challenges and open questions. In *Proceedings of the 4th Central-European Workshop on Services and their Composition, ZEUS*, volume 847, pages 50–57. Citeseer, 2012.

[77] Francesco De Angelis, Daniele Fanì, and Andrea Polini. Partes: A test generation strategy for choreography participants. In Hong Zhu, Henry Muccini, and Zhenyu Chen, editors, *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, pages 26–32. IEEE Computer Society, 2013. doi: 10.1109/IWAST.2013.6595787.

[78] Midhat Ali, Francesco De Angelis, Daniele Fanì, Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. An extensible framework for online testing of choreographed services. *IEEE Computer*, 47(2):23–29, 2014. doi: 10.1109/MC.2013.407.

[79] Lei Zhou, Jing Ping, Hao Xiao, Zheng Wang, Geguang Pu, and Zuohua Ding. Automatically testing web services choreography with assertions. In *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2010. doi: 10.1007/978-3-642-16901-4_11.

[80] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. doi: 10.1007/978-3-540-78917-8_1.

[81] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Softw. Concepts Tools*, 17(3):103–120, 1996.

[82] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984. doi: 10.1016/0304-3975(84)90113-0.

[83] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 748–761, 2017. ISSN 07308566. doi: 10.1145/3009837.3009847.

[84] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In *the 40th International Conference*, pages 1137–1148. ACM Press, sep 2018. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180157.

[85] Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg, and Yaroslav Usenko. From μCRL to mCRL2. Motivation and Outline. *Electronic Notes in Theoretical Computer Science*, 2006. ISSN 15710661. doi: 10.1016/j.entcs.2005.12.101.

[86] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local verification of global protocols. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8174 LNCS: 358–363, 2013. ISSN 03029743. doi: 10.1007/978-3-642-40787-1_25.

[87] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. *ACM SIGPLAN Notices*, 2016. ISSN 15232867. doi: 10.1145/2837614.2837634.

[88] David Harel and PS Thiagarajan. Message sequence charts. In *UML for Real*, pages 77–105. Springer, 2003. doi: 10.1007/0-306-48738-1_4.

[89] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: A survey. *Software and Systems Modeling*, 10(4):489–514, 2011. ISSN 16191366. doi: 10.1007/s10270-010-0157-9.

[90] David Harel and Rami Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software & Systems Modeling*, 2(2):82–107, 2003. doi: 10.1007/s10270-002-0015-5.

[91] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001. doi: 10.1023/A:1011227529550.

[92] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based api tester for event-driven systems. In *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings 9*, pages 112–128. Springer, 2013.

[93] Kwang Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, pages 86–91, 1993. doi: 10.1145/157485.164585.

[94] Cyrille Artho, Kazuaki Banzai, Quentin Gros, Guillaume Rousset, Lei Ma, Takashi Kitamura, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Model-based testing of Apache ZooKeeper: Fundamental API usage and watchers. *Software Testing, Verification and Reliability*, 30(7-8):e1720, 2020. ISSN 1099-1689. doi: 10.1002/stvr.1720.

[95] Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. Building Reliable Cloud Services Using P# (Experience Report). *arXiv:2002.04903 [cs]*, February 2020.

[96] Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. Building Reliable Cloud Services Using Coyote Actors. In *SoCC*, November 2021.

[97] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. `https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/`.

[98] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012. doi: 10.1145/2093548.2093564.

[99] Laura Bocchi, Hernán C. Melgratti, and Emilio Tuosto. Resolving non-determinism in choreographies. In *ESOP*, LNCS, pages 493–512. Springer, 2014. doi: 10.1007/978-3-642-54833-8_26.

[100] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013. doi: 10.1145/2429069.2429101.

[101] Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *Journal of Logical and Algebraic Methods in Programming*, 108:69–89, sep 2019. ISSN 23522208. doi: 10.1016/j.jlamp.2019.06.003.

[102] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *TOSEM*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[103] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*, pages 483–484. Springer, 2002.

[104] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. A tool for choreography-based analysis of message-passing software. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, chapter 6, pages 125–146. River Publisher, 2017.

[105] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 643–653. ACM, 2014. doi: 10.1145/2635868.2635920.

[106] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 830–840. ACM, 2019. doi: 10.1145/3338906.3338945.

[107] Martin Fowler. Eradicating Non-Determinism in Tests. `https://martinfowler.com/articles/nonDeterminism.html`. Accessed: 2021-04-13.

[108] Jukka Lehtosalo, G v Rossum, Ivan Levkivskyi, Michael J Sullivan, David Fisher, Greg Price, Michael Lee, N Seyfer, R Barton, S Ilinskiy, et al. Mypy-optional static typing for python, 2017. `http://mypy-lang.org/`. Accessed: 2020-10-30.

[109] Pyright - static type checker for python. `https://microsoft.github.io/pyright/`.

[110] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, August 2015. Association for Computing Machinery. ISBN 978-1-4503-3810-3. doi: 10.1145/2808098.2808100.

[111] Riccardo Pucella and Jesse A Tov. Haskell Session Types with (Almost) No Class. pages 25–36, 2008. doi: 10.1145/1411286.1411290.

[112] Antonia Bertolino and Eda Marchetti. A brief essay on software testing. *Software Engineering, 3rd edn. Development process*, 1:393–411, 2005.

[113] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232. ACM, 2015. doi: 10.1145/2676726.2676964.

[114] Wil M. P. van der Aalst. Process mining: Overview and opportunities. *ACM Trans. Manag. Inf. Syst.*, 3(2):7:1–7:17, 2012. doi: 10.1145/2229156.2229157.

[115] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002. doi: 10.1006/inco.2001.3080.

[116] Rocco De Nicola and Hernán C. Melgratti. Multiparty testing preorders. In *Trustworthy Global Computing*, pages 16–31, 2015. doi: 10.1007/978-3-319-28766-9_2.

[117] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. doi: 10.1145/267580.267590.

[118] A.J. Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99) (Cat. No.PR00434)*, pages 119–129, October 1999. doi: 10.1109/ICECCS.1999.802856.

[119] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472, pages 281–291. Springer Berlin Heidelberg, Berlin, Heidelberg, oct 2005. ISBN 978-3-540-26278-7 978-3-540-32037-1. doi: 10.1007/11498490_13.

[120] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP*, LNCS, pages 194–213. Springer, 2012. doi: 10.1007/978-3-642-28869-2_10.

[121] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino, and Andrea Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007. doi: 10.1002/stvr.357.

[122] Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019. doi: 10.1007/978-3-030-25540-4_6.

[123] Franco Barbanera, Ugo de'Liguoro, and Rolf Hennicker. Global types for open systems. In Massimo Bartoletti and Sophia Knight, editors, *Proceedings 11th Interaction and Concurrency Experience, ICE 2018, Madrid, Spain, June 20-21, 2018*, volume 279 of *EPTCS*, pages 4–20, 2018. doi: 10.4204/EPTCS.279.4.

[124] Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. On the k-synchronizability of systems. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2020. doi: 10.1007/978-3-030-45231-5_9.

[125] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013. doi: 10.1007/978-3-642-39212-2_18.

[126] Roly Perera, Julien Lange, and Simon J Gay. Multiparty Compatibility for Concurrent Objects. *Electronic Proceedings in Theoretical Computer Science*, 211: 73–82, oct 2016. ISSN 2075-2180. doi: 10.4204/EPTCS.211.8.