An Approach for the Verification and Synthesis of Complete Test Generation Algorithms for Finite State Machines

Dissertation zur Erlangung des
Doktorgrades der Ingenieurswissenschaften
(Dr.-Ing.)

vorgelegt von
Robert Sachtleben

im Fachbereich 3 (Mathematik und Informatik)
der Universität Bremen
im Mai 2022

Datum des Promotionskolloquiums: 20. Juli 2022

Gutachter:
Prof. Dr. Jan Peleska (Universität Bremen)
Prof. Dr. Burkhart Wolff (Universität Paris-Saclay)

Zusammenfassung

Vollständige Testsuites sind von besonderer Bedeutung im modellbasierten Testen, da sie unter eindeutig spezifizierten Annahmen eine hohe garantierte Teststärke aufweisen. Insbesondere für Systemspezifikationen auf Grundlage endlicher Zustandsmaschinen wurden daher zahlreiche Teststrategien entworfen, die solche vollständigen Testsuites generieren. Die Vielzahl und teils hohe Komplexität dieser Strategien, ihrer Implementierungen, und der zum Nachweis ihrer Vollständigkeit erbrachten Beweise erschweren jedoch manuelle Korrektheitsprüfungen. Uneindeutigkeiten oder nur implizit getroffene Annahmen in textuellen Beschreibungen und Beweisen können zudem leicht zu unvollständigen Implementierungen führen.

Die vorliegende Dissertation beschreibt einen alternativen Ansatz für die Verifikation und Implementierung von Teststrategien für die Erzeugung von vollständigen Testsuites für Sprachäquivalenz endlicher Zustandsmaschinen mithilfe des interaktiven Theorembeweisers Isabelle.

Im ersten Teil der Arbeit werden Gemeinsamkeiten und Unterschiede der Teilschritte der Teststrategien identifiziert. Daraus wird ein Framework in Form einer Funktion höherer Ordnung abgeleitet, das es erlaubt, die untersuchten Strategien als spezifische Verwendungen dieses Frameworks aufzufassen. Viele Implementierungen einzelner Parameter dieses Frameworks werden zwischen verschiedenen Strategien wiederverwendet. Strategien, die bisher nur auf deterministischen Zustandsmaschinen definiert waren, werden dabei auf potentiell nichtdeterministische Systeme verallgemeinert. Zwei weitere Frameworks werden hinzugefügt, die alternative Vollständigkeitsbeweise unterstützen.

Der zweite Teil der Arbeit definiert endliche Zustandsmaschinen, ihre Eigenschaften und Operationen, und schließlich die Frameworks in Isabelle/HOL und beweist ihre Korrektheit bzw. Vollständigkeit. Vollständigkeitsbeweise für die Frameworks werden hierbei durch Spezifikation geeigneter Vor- und Nachbedingungen von konkreten Implementierungen ihrer Parameter entkoppelt. Dies erlaubt die Wiederverwendung von Beweisen zwischen ähnlichen Strategien und vereinfacht so Vollständigkeitsbeweise für neue Varianten der bereits behandelten Strategien.

Die Definitionen in Isabelle/HOL werden schließlich im dritten Teil der Arbeit verwendet, um beweisbar korrekte, ausführbare Implementierungen der Teststrategien automatisch zu generieren. Es wird gezeigt, dass diese für bestimmte Eingaben eine mit einer manuell entwickelten C++ Bibliothek vergleichbare Performanz aufweisen.

Abstract

Complete test suites are of special interest in the field of model-based testing, as they guarantee high fault detection capabilities under well-specified assumptions. In particular for specifications given as finite state machines, many test strategies have been developed that generate complete test suites. The variety and sometimes high complexity of such strategies, their implementations, and their completeness proofs impose difficulties upon manual verification. Furthermore, ambiguities or only implicitly specified assumptions in natural language descriptions and proofs easily lead to implementations that do not generate complete test suites.

The present dissertation proposes an alternative approach to the verification and implementation of test strategies that generate complete test suites for testing for language-equivalence on finite state machines, that employs the interactive theorem prover Isabelle.

In the first part of this work, similarities and differences between steps performed by the considered test strategies are identified. From these, a framework is derived as a higher order function capable of representing all considered strategies as concrete applications of it. Several implementations of parameters of this framework are shared between multiple strategies. Strategies that have previously only been defined on deterministic state machines are generalised to be applicable to potentially nondeterminstic state machines. Two additional frameworks are developed to support alternative completeness proofs.

The second part of this work defines finite state machines, their properties and operations, as well as the frameworks in Isabelle/HOL and respectively proves them correct and complete. Completeness proofs over frameworks are decoupled from concrete implementations of their parameters by suitable pre- and post-conditions. This approach enables the reuse of proofs between similar strategies and thus simplifies completeness proofs for new variations on already handled strategies.

Finally, the definitions in Isabelle/HOL are employed in the third part of this work to automatically generate provably correct implementations of the considered test strategies. It is shown that these exhibit comparable performance to a manually developed C++ library for certain inputs.

Acknowledgement

I am indebted to my advisor Professor Jan Peleska for suggesting that I combine my prior interests in proof assistants and complete test strategies, as well as for supporting and guiding me in the exploration of their interaction and application, resulting in the present dissertation.

I would furthermore like to thank my colleagues at the research group *Operating Systems, Distributed Systems* for many fruitful discussions. In particular, I owe thanks to Professor Wen-ling Huang for clarifying and improving several proofs on the SPY and SPYH-Methods, and I am grateful to Niklas Krafczyk for enduring many conversations on Isabelle/HOL.

Finally, I wish to thank my good friend Nico Heller for many dialogues on performance and optimisation, as well as his support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

In the field of testing safety-critical systems such as controllers of airbags, aircraft engines, or medical equipment by deriving test suites from a reference model, known as *model-based testing* (MBT), so-called *complete* test strategies are of high interest, as they guarantee fault coverage over well-specified fault domains under explicit assumptions. Many such strategies have been developed for models that are represented as a *finite state machine* (FSM), which describes the behaviour of a system as a finite collection of states that react to inputs by producing an output an updating the current state of the system. Via techniques such as the construction of equivalence classes, these strategies furthermore serve as foundations of complete strategies for more elaborate modelling formalisms, as detailed in [48, 50, 83].

In order for them to be applicable in the certification of safety-critical systems, it is crucial that the completeness claims of test strategies are established convincingly. Due to the large number of distinct test strategies and variations thereof, as well as the intricacy of some completeness proofs, however, it cannot be expected that each such proof is thoroughly manually checked by many members of the testing community. This view is supported by my previous work (see [96]), which uncovered an ambiguity in the natural language description of a test strategy that could lead to implementations that do not possess the claimed fault detection capabilities.

I thus advocate the use of proof assistant tools to mechanically verify completeness proofs. These may then be presented to certification authorities as artefacts that are easier to verify than proofs in natural language, thus reducing the effort required to establish completeness. In many state-of-the-art theorem provers and proof assistants such as Isabelle or Coq, it is furthermore possible to derive executable implementations of test strategies from definitions developed to describe these strategies in a mechanised proof (see [42, 68]). That is, a suitable mechanised proof does not only prove a test strategy complete but

also provides a provably correct implementation of the strategy. The feasibility of this approach has been demonstrated in my previous work [95].

1.2 Contributions

The present work develops mechanised completeness proofs for a collection of test strategies that generate test suites for the *language-equivalence* conformance relation over FSMs, where a *system under test* conforms to a reference model if it exhibits the same behaviour with respect to inputs and outputs. This mechanisation effort is performed using the well-known Isabelle proof assistant[1] and encompasses three main contributions:

1. *Generic Frameworks* – I show that the considered test strategies may all be viewed as concrete implementations of a single generic framework. That is, I develop a higher order function such that all considered test strategies may be implemented by providing suitable arguments to this function. The decomposition of test strategies into groups of smaller functions furthermore simplifies the later implementation of additional test strategies, as these may reuse parts of the considered strategies. I develop two additional frameworks for subsets of the considered strategies that can be proven complete by alternative arguments.

2. *Extensible mechanised proofs* – I establish completeness of the considered strategies in two steps. First, I prove complete all applications of the framework, assuming that the functions passed to it satisfy certain properties. Thereafter, I show that the implementations of the strategies derived in the previous contribution provide such sufficient functions as arguments. This approach significantly simplifies proving complete a test strategy that reuses parts of previously considered strategies, as the properties of these parts have already been established.

3. *Trustworthy test tools* – From the definitions in the mechanised proof I generate provably correct implementations of the considered test strategies, which I employ in a simple tool set suitable for generating test suites for a given reference model and applying them to a system under test.

The mechanised formalisation of test strategies on finite state machines additionally require the formalisation of finite state machines and various properties of and operations on these. The resulting library of definitions and functions constitutes a further small contribution of this work.

Finally, some of the test strategies considered here have previously been defined only on deterministic reference models and systems under test. These I have generalised to be applicable to nondeterministic systems.

[1] https://isabelle.in.tum.de

1.3 Related Work

An overview of model-based testing and approaches to test suite generation is presented in [2, 113, 114]. Examples of applications and benefits with respect to avionic systems are discussed in [80, 81].

For FSMs, a large number of test strategies for various conformance relations has been developed. With respect to testing for *language-equivalence*, the development of strategies ranging from the W-Method of [23, 115] to the SPYH-Method of [106] is described in detail in Chapter 5. Other conformance relations on FSMs such as *reduction* (see [44, 87, 88, 91]), *quasi-equivalence* and *quasi-reduction* (see [43, 45, 87, 88, 89]) or *strong reduction* (see [97]) have also been proposed and provided with complete test strategies. Furthermore, complete strategies have been developed for more expressive formalisms such as labelled transition systems, which give rise to more intricate conformance relations. For an overview see [111]. For other domains such as reactive IO-state-transition systems (RIOSTS), it has been shown in [48] that complete test strategies may be derived from complete strategies for FSMs after partitioning states and inputs into equivalence classes.

An overview of the Isabelle proof assistant employed in the present work is given in [78, 120]. Its human readable *Isar* language is described in [121] and the *Sledgehammer* automated proof tool is introduced and evaluated in [6, 8, 11, 77]. A large number of results already formalised with Isabelle is available at the *Archive of Formal Proofs*[2], which has been analysed in [9]. Isabelle has been employed for several large-scale proof efforts, including the verification of the seL4 microkernel (see [57, 58]) and the pervasive system-level verification performed in the Verisoft project (see [1]). In [16], Bourke et al. discuss the challenges encountered in managing such large-scale proofs. Further examples and capabilities of Isabelle are discussed in Part III. A survey of the literature regarding the engineering of proofs is presented in [94].

The application of proof assistants to testing has, to my best knowledge, first been advocated by Brucker and Wolff in [18]. In [20], they present an integrated testing framework that employs Isabelle/HOL at its core, allowing for test strategy[3] elaboration, fault coverage proof, test case and test data generation in the same tool. The framework is extended by the same authors in [19] by generalisations of various types of automata, including mealy machines, and capabilities to refine test cases for conformance relations on such automata. The authors present several cases of mechanised proofs establishing the completeness of test strategies. These do not include the strategies considered in this work.

The general approach to model-based testing I present in this work and have presented previously with Hierons, Huang and Peleska in [95, 96] furthermore contrasts to the one advocated in [20] and [19], in that we favour separate specialised tools for strategy elaboration (Isabelle/HOL), modelling of the reference implementation (FSM and SysML modelling tools), and test case and test data

[2] https://www.isa-afp.org

[3] A strategy is called a *test theorem* in [20].

generation (RT-Tester [81] with SMT solvers [84])[4]. I furthermore agree with [5] that the use of SMT solvers in testing requires less specialised expertise than the interactive handling of proof assistants, since the internal application of SMT solving does not require direct interactions with users.

1.4 Reference to Online Resources

The Isabelle/HOL formalisation described in this work is publicly available for download at `https://bitbucket.org/RobertSachtleben/an-approach-for-the-verification-and-synthesis-of-complete`. This repository also contains the tool set using code generated from the formalisation, as well as evaluation data for statistical experiments performed using these tools. The content of the repository is further detailed in the contained README file.

In addition to the repository, the theory files constituting the formalisation developed in Isabelle will be submitted to the Archive of Formal Proofs after this dissertation has been accepted.

1.5 Outline of this Document

The structure of this dissertation follows the main contributions. First, Part I introduces finite state machines and defines relevant properties and operations on these, followed by a short introduction of model-based testing with respect to reference models specified as FSMs. Next, Part II discusses complete strategies for testing the language-equivalence conformance relation, highlighting the sufficient conditions they employ to establish completeness, and thereupon develops generic frameworks suitable to describe some or all of the considered strategies. The mechanised formalisation of these frameworks, strategies, and completeness proofs is presented in Part III, which also elaborates on the formalisation of definitions and operations discussed in Part I. Thereafter, Part IV describes how this formalisation is employed to generate provably correct implementations of the considered test strategies, and how these are embedded into a small tool set to interact with practical systems. Finally, Part V summarises these results and discusses possible future research topics. Appendices A to E provide supplementary information.

[4] This last step requires separate tools only if the reference model is not an FSM and requires additional translation or abstraction of test cases as described in, for example, [48].

Part I

Background

Chapter 2

Basic Definitions

This chapter introduces various definitions related to finite state machines as employed in subsequent chapters, following standard definitions as used in, for example, [43, 97, 103, 108].

First, Section 2.1 introduces basic definitions and operations on sequences an sets thereof. Next, Section 2.2 defines finite state machines, paths, and languages. Thereafter, Sections 2.3 and 2.4 discuss key steps performed by most test strategies on FSMs: reaching and distinguishing states. Finally, Section 2.5 defines the notion of converging test cases as introduced in [103], which enables some test strategies to reduce the size of generated test suites.

2.1 Sequences

Throughout this work, a *sequence* or *trace* $\bar{x} = x_1.x_2.x_3 \dots x_k$ is a finite list of elements of some set X. The length of sequence $\bar{x}$ is denoted $|\bar{x}| = k$. The *empty* sequence is denoted ϵ and is of length 0, containing no elements. Concatenation of sequences $\bar{x}, \bar{x}'$ is denoted $\bar{x}.\bar{x}'$ or simply $\bar{x}\bar{x}'$, where $\bar{x}'$ is said to *extend* $\bar{x}$. The *prefixes* of $\bar{x}$, denoted $pref(\bar{x})$, are all $\bar{x}'$ such that there exists an extension $\bar{x}''$ such that $\bar{x} = \bar{x}'.\bar{x}''$. The empty sequence is prefix of all sequences. A prefix of $\bar{x}$ is *proper* if it is shorter than $\bar{x}$.

In describing test strategies, it is often necessary to consider sets of sequences of certain lengths over a fixed set of elements. Let A be some set and let $i \in \mathbb{N}$. Then A^i denotes the set of all sequences of length i that consist only of elements in A. Furthermore, A^* denotes the set of all finite sequences over A, while $A^{\leq i}$

denotes the set of all sequences over A of length at most i. That is,

$$A^i := \begin{cases} \{\epsilon\} & \text{if } i = 0 \\ A & \text{if } i = 1 \\ \{a_1.a_2.\ldots.a_{i-1}.a_i \mid \forall 1 \leq j \leq i.\, a_j \in A\} & \text{else} \end{cases}$$

$$A^{\leq i} := \bigcup_{j=0}^{i} A^j$$

$$A^* := \bigcup_{j=0}^{\infty} A^j$$

The *prefix completion* of a set of sequences X, denoted $\mathit{pref}(X)$, is the set of all prefixes of all sequences in X.

$$\mathit{pref}(X) := \bigcup_{\bar{x} \in X} \mathit{pref}(\bar{x})$$

Similarly, the notion of extension is lifted to sets of sequences, denoting as $A.B$ the extension of every sequence in A with every sequence in B. If one of the sets is empty, $A.B$ is defined as the other set. That is,

$$A.B := \begin{cases} A & \text{if } B = \emptyset \\ B & \text{if } A = \emptyset \\ \{\alpha.\beta \mid \alpha \in A \wedge \beta \in B\} & \text{else} \end{cases}$$

2.2 Finite State Machines

Finite state machines model the behaviour of a system consisting of *states* and *transitions* between states. Transitions are labelled with input-output (IO) pairs and represent possible responses of a system in a given state to a given input by producing some output and changing the current state of the system to the target state of the transition.

Definition 2.2.1 (FSM)**.** A *finite state machine* (FSM) is a 5-tuple

$$M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$$

consisting of a finite set of *states* Q, an *initial state* $q_0 \in Q$, a finite set Σ_I of *input symbols*, called *input alphabet*, a finite set Σ_O of *output symbols*, called *output alphabet*, and a finite set $h_M \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$ of *transitions*. Each transition (q, x, y, q') in turn consists of a *source* state q, an input x, an output y, and a *target* state q'.

The *size* of M, denoted $|M|$, is the number $|Q|$ of states of M. ⊣

This definition follows the literature [28, 44, 47, 72, 89] in introducing FSMs as *Mealy automata* (see [73]). That is, finite state machines as employed in the

present work do not exhibit *accepting* or *rejecting* states that terminate their behaviour. By also distinguishing between inputs and outputs, they are thus suitable to model reactive control systems.

FSMs may be represented visually by depicting states as nodes in a graph and each transition (q, x, y, q') as an edge from state q to q' labelled with x/y. In the following, I collapse parallel edges (q, x, y, q'), (q, x', y', q') into single edges labelled with both x/y and x'/y'. Finally, the initial state is indicated by an incoming edge without source state or label. For example, Fig. 2.1 is a graphical representation of $M_2 = (Q, q_0, \Sigma_I, \Sigma_O, h)$ where

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3\} \\ \Sigma_I &= \{a, b\} \\ \Sigma_O &= \{1, 2, 3\} \\ h &= \{(q_0, a, 0, q_1), (q_0, a, 1, q_2), (q_0, b, 2, q_2), \\ &\quad (q_1, a, 1, q_0), (q_1, b, 0, q_1), (q_1, b, 1, q_1), \\ &\quad (q_2, a, 2, q_1), (q_2, b, 1, q_3), \\ &\quad (q_3, a, 0, q_1)\} \end{aligned}$$

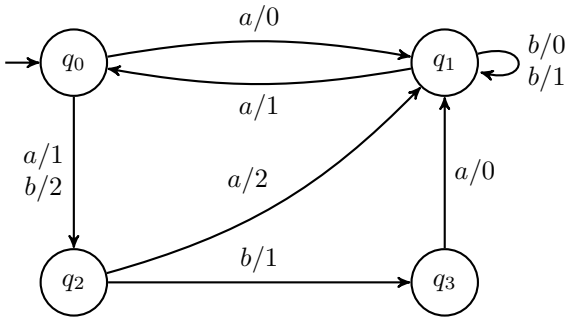


Figure 2.1: FSM M_2.

In the following, I often employ only a graphical representation to specify some FSM, omitting explicit enumeration of the states, alphabets, and transitions. In such cases, the graphical representation comprises all elements. That is, graphical representations show all states, transitions, and input or output symbols of the represented FSM directly.

The behaviour of an FSM beyond single transitions is represented by the paths between its states.

Definition 2.2.2 (Path)**.** A finite sequence

$$p = (q_1, x_1, y_1, q_1').(q_2, x_2, y_2, q_2') \ldots (q_{k-1}, x_{k-1}, y_{k-1}, q_{k-1}').(q_k, x_k, y_k, q_k')$$

of transitions is a *path* in FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ *from* state q_1 *to* state q_k' if for all $1 \leq i \leq k$ it holds that $(q_i, x_i, y_i, q_i') \in h_M$ and also $q_j' = q_{j+1}$ for all $j \leq i < k$. The empty sequence ϵ is a path from any state of M to itself. $\dashv$

The *IO-projection* of path p is the trace of input-output pairs resulting from mapping function $(q, x, y, q') \mapsto (x/y)$ over p. The set of all IO-projections on paths constitutes the *language* of an FSM.

Definition 2.2.3 (Language)**.** The *language* $\mathcal{L}_M(q)$ of state q in FSM M is the set of all IO-traces $(x_1/y_1).(x_2/y_2)\ldots(x_k/y_k)$ such that there exists a path $(q, x_1, y_1, q'_1).(q_2, x_2, y_2, q'_2)\ldots(q_k, x_k, y_k, q'_k)$ from q in M. The language of M itself, denoted $\mathcal{L}(M)$, is the language of its initial state. ⊣

For example, $(a/1).(a/1) \in \mathcal{L}_{M_2}(q_1)$ but $(a/1).(a/1) \notin \mathcal{L}_{M_2}(q_0) = \mathcal{L}(M_2)$.

In the following, I occasionally write an IO-trace $(x_1/y_1).(x_2/y_2)\ldots(x_k/y_k)$ as $x_1x_2\ldots x_k/y_1y_2\ldots y_k$, where $x_1x_2\ldots x_k$ and $y_1y_2\ldots y_k$ are respectively denoted the *input* and *output portion* of the trace. Furthermore, in the following, $\bar{x}$ always denotes an input sequence, whereas $\bar{x}/\bar{y}$ denotes IO-traces. In cases where it is not necessary to individually name the elements or portions, I also employ Greek letters as names of traces or input sequences. Finally, in order to more closely resemble certain proofs on paper in the literature, traces are sometimes also named u, v, or w. The projection of a set of IO-traces A over input alphabet Σ_I to input sequences is denoted $A|_{\Sigma_I}$. For example, $\{x_1x_2/y_1y_2, x'_1/y'_1\}|_{\Sigma_I} = \{x_1x_2, x'_1\}$.

By the composition of their transitions, FSMs may be classified over a variety of properties:

Definition 2.2.4 (Basic properties)**.** FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ is

- *deterministic* if each state has at most one transition for each input:

$$\forall (q_1, x_1, y_1, q'_1), (q_2, x_2, y_2, q'_2) \in h_M.\\ (q_1 = q_2 \wedge x_1 = x_2) \longrightarrow (y_1 = y_2 \wedge q'_1 = q'_2)$$

- *observable* if each state has at most one emanating transition for each input-output pair:

$$\forall (q_1, x_1, y_1, q'_1), (q_2, x_2, y_2, q'_2) \in h_M.\\ (q_1 = q_2 \wedge x_1 = x_2 \wedge y_1 = y_2) \longrightarrow q'_1 = q'_2$$

- *completely specified* (or *complete*) if each state has at least one transition for each input:

$$\forall q \in Q.\, \forall x \in \Sigma_I.\, \exists y, q'.\, (q, x, y, q') \in h_M$$

- *partial* if at least one state has no transitions for at least one input:

$$\exists q \in Q.\, \exists x \in \Sigma_I.\, \nexists y, q'.\, (q, x, y, q') \in h_M$$

- *minimal* if no two states of M exhibit the same language:

$$\forall q_1, q_2 \in Q.\, q_1 \neq q_2 \longrightarrow \mathcal{L}_M(q_1) \neq \mathcal{L}_M(q_2)$$ ⊣

In the following, I abbreviate deterministic FSMs as DFSMs and observable FSMs as OFSMs.

FSM M_2 of Fig. 2.1 is not deterministic due to multiple transitions emanating from q_0 for input a, but it is observable, as no two transitions emanating from the same state exhibit the same IO-pair. Furthermore, it is partial, since no transition for input b emanates from q_3. Finally, it is minimal, as no pair of distinct states of M_2 exhibits the same set of responses to input a.

Every non-observable FSM M can be transformed into an observable FSM M' such that $\mathcal{L}(M) = \mathcal{L}(M')$. Similarly, every non-minimal FSM may be transformed into a minimal FSM of the same language by an algorithm that preserves observability. Algorithms for these transformations are discussed and formalised in Sections 8.4 and 8.5, respectively. When considering conformance relations that depend solely on the language of an FSM, it is thus no restriction to assume observability and minimality.

2.3 Reaching States

A transition (q, x, y, q') only affects the language of an FSM M if state q is target of some path from the initial state of M, since otherwise the transitions is not contained in any path from the initial state. The behaviour of an FSM thus depends only on states that are *reached* by some path from the initial state.

Definition 2.3.1 (Reachability)**.** A state q of FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ is *reachable* if there exists a path from q_0 to q in M. A trace $\alpha \in \mathcal{L}(M)$ *reaches* state q' if there exists a path in M from q_0 to q' such that α is the IO-projection of this path. ⊣

Similarly, a path p from a state q *visits* state q' if there exists a prefix of p whose target is p', and this notion is extended to IO-traces analogously to reachability. An FSM is called *prime* if it is minimal, observable, and all of its states are reachable. As non-reachable states do not affect the language of an FSM, their removal is language-preserving.

Note that in observable FSMs M, if p_1 and p_2 are paths from the same state and the IO-projections of p_1 and p_2 coincide, then $p_1 = p_2$, as by definition of observability they must exhibit the same intermediate states. Thus, if $\alpha \in \mathcal{L}_M(q)$, then there exists a unique path whose IO-projection is α. Therefore, reachability by paths coincides with reachability via IO-traces in OFSMs. I introduce the following function to denote the state reached by an IO-trace in an OFSM:

Definition 2.3.2 (after)**.** Function *after* determines a state reached in an observable FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ via trace α applied to some $q \in Q$:

$$q\text{-after-}\alpha = \begin{cases} q & \text{if } \alpha = \epsilon \\ q'\text{-after-}\alpha' & \text{if } \alpha = (x/y).\alpha' \text{ and } (q, x, y, q') \in h_M \\ \textit{undefined} & \text{else} \end{cases}$$

I write M-after-α to denote q_0-after-α. ⊣

For example, in FSM M_2 of Fig. 2.1, q_0-after-$aab/120 = q_2$-after-$ab/20 = q_1$-after-$b/0 = q_1$ and hence $aab/120$ reaches q_1.

I extend function *after* to input sequences by denoting as q-after-$\bar{x}$ the union of all sets q-after-$\bar{x}/\bar{y}$ for traces $\bar{x}/\bar{y}$ in the language of state q. Note that in the case of completely-specified DFSMs, the target state reached from some q by an input sequence $\bar{x}$ is uniquely determined. In this case, I employ q-after-$\bar{x}$ to donate this state instead of the singleton state containing it.

Traces reaching states of an FSM are often collected in so-called *state covers*, constituting the first step of most test strategies discussed in the present work (see Chapter 5).

Definition 2.3.3 (State cover)**.** A *state cover* V of an observable FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ is a set of traces such that $\epsilon \in V$ and for each state $q \in Q$ there exists in V a trace v_q that reaches q in M. That is, if

$$\forall q \in Q. \exists v_q \in V. v_q \in \mathcal{L}(M) \wedge M\text{-after-}v_q = q$$

holds, then V is a state cover of M. ⊣

For example, set $pref(aba/110) = \{\epsilon, a/1, ab/11, aab/110\}$ constitutes a state cover of M_2, as $aba/110$ visits all states of M_2.

For each OFSM M with states $Q = \{q_0, \ldots, q_n\}$ and each state cover V of M, there must thus exist $\{v_{q_0}, \ldots, v_{q_n}\} \subseteq V$ such that for each $q \in Q$, v_q reaches q. Note that in deterministic FSMs each input sequence $\bar{x}$ induces at most one trace $\bar{x}/\bar{y} \in \mathcal{L}(M)$ and hence the output portions of traces in a state cover can be omitted for DFSMs. In the following, I write $\bar{v}_q$ to denote the input portion of v_q.

Transition covers extend the notion of state covers by requiring traversal of all transitions in an FSM:

Definition 2.3.4 (Transition cover)**.** A *transition cover* TC of an OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ is a set of traces such that for each state $q \in Q$ and each IO-pair $x/y \in \Sigma_I \times \Sigma_O$ there exists in TC a trace $\alpha.(x/y).\beta$ such that α reaches q in M. That is, if

$$\forall q \in Q, x \in \Sigma_I, y \in \Sigma_O. \exists \alpha. \alpha \in \mathcal{L}(M) \wedge M\text{-after-}\alpha = q \wedge \alpha.(x/y) \in TC$$

holds, then TC is a transition cover of M[1]. ⊣

[1]The definition presented here generalises to IO-traces the definition of transition covers presented in [103], which considers only completely specified DFSMs. Contrary to their name, *transition* covers do not explicitly depend on the transitions of an FSM and instead require application of all possible IO-pairs (or inputs in case of [103]) after reaching traces of all states. Also note that transition covers are thus defined only on FSMs without unreachable states.

2.4 Distinguishing States

In testing against FSM specifications, it is often required to decide whether the current state of the system under test conforms to at most one of a pair of states q, q' of the reference model. A basic instrument in performing such checks are so-called *distinguishing* traces α such that containment of α in the language of the current state of the SUT implies non-conformance with q, whereas non-containment of α implies non-conformance with q', or vice versa.

Definition 2.4.1 (Distinguishability, Δ)**.** Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be an FSM containing states q, q'. Then $\Delta_M(q, q') \subseteq (\Sigma_I \times \Sigma_O)^*$ denotes the set of all traces *distinguishing* q and q', defined as follows:

$$\Delta_M(q, q') := (\mathcal{L}_M(q) \setminus \mathcal{L}_M(q')) \cup (\mathcal{L}_M(q') \setminus \mathcal{L}_M(q))$$

That is, a trace α distinguishes q and q' if it is contained in the language of only one of these states. ⊣

If the reference model M is unambiguous from the context, I drop the subscript of Δ_M. An input sequence distinguishes states q, q' if it is the input portion of an IO-trace distinguishing the states.

In FSM M_2 of Fig. 2.1, input a constitutes a distinguishing sequence for all pairs of distinct states, since $a/0$ distinguishes (q_0, q_1), (q_0, q_2), (q_1, q_3), and (q_2, q_3), while $a/1$ distinguishes (q_0, q_3) and (q_1, q_2). Section 8.6 discusses a verified function to compute distinguishing traces and provides further examples.

Distinguishing traces are also employed to check whether a pair of traces reaches distinct states:

Definition 2.4.2 (Separability)**.** Consider $\alpha, \beta \in \mathcal{L}(M)$ for minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ such that M-after-$\alpha \neq M$-after-β. Furthermore let $TS \subseteq (\Sigma_I \times \Sigma_O)^*$. Then traces α and β are *TS-separable* if there exists some $\omega \in \Delta_M(M\text{-after-}\alpha, M\text{-after-}\beta)$ such that $\{\alpha.\omega, \beta.\omega\} \subseteq pref(TS)$. ⊣

That is, α and β are TS-separable if TS extends both with the some trace ω distinguishing their reached states in M. In this case, ω *separates* α and β. The notion of separability extends naturally to sets of traces:

Definition 2.4.3 (Separability of sets)**.** Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal OFSM and let A, $TS \subseteq (\Sigma_I \times \Sigma_O)^*$. Then A is TS-separable if for all $\alpha, \beta \in \mathcal{L}(M)$ such that M-after-$\alpha \neq M$-after-β, α and β are TS-separable. ⊣

The choice of separating traces is a major aspect distinguishing the various test strategies discussed in Chapter 5. Some strategies compute sets of separating traces once and ensure separability of some set of traces A by extending traces in A with one or more of the following constructions. The most important such sets are defined as follows.

Definition 2.4.4 (Characterisation set)**.** A set $W \subseteq (\Sigma_I \times \Sigma_O)^*$ is a *characterisation set* of a minimal FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ if the following holds

$$\forall q, q' \in Q.\, q \neq q' \longrightarrow pref(W) \cap \Delta_M(q, q') \neq \emptyset$$

That is, a characterisation set contains a distinguishing trace for every pair of distinct states of M. ⊣

Definition 2.4.5 (State identifier)**.** A set $W_q \subseteq (\Sigma_I \times \Sigma_O)^*$ is a *state identifier* for state q of a minimal FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ if the following holds

$$\forall q' \in Q.\, q \neq q' \longrightarrow pref(W_q) \cap \Delta_M(q, q') \neq \emptyset$$

That is, a state identifier of q contains sufficient traces to distinguish q from all other states of M. ⊣

Definition 2.4.6 (Harmonised state identifiers)**.** Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be an FSM and suppose that $Q = \{q_1, q_2, \ldots, q_k\}$. Let $H = \{H_1, H_2, \ldots, H_k\}$ such that for each $1 \leq i \leq k$, H_i is a state identifier of q_i. Then H is a family of *harmonised* state identifiers if for every pair of distinct states it holds that their corresponding state identifiers share a trace distinguishing the state pair – that is, if the following holds:

$$\forall 1 \leq i < j \leq k.\, pref(H_i) \cap pref(H_j) \cap \Delta_M(q_i, q_j) \neq \emptyset$$ ⊣

2.5 Convergence

In [103], Simão et al. have shown that if it has been established during testing that some traces α, β reach the same state in both the reference model and the system under test, then certain steps of test strategies such as extending α with a distinguishing trace γ may instead also be performed on β while preserving the property checked in doing so. This approach allows minimisation of test suites by providing more options in the distribution of extensions, as introduced in [103, 106] and discussed further in Subsections 5.1.6 and 5.1.7.

In [103] and [106], only complete DFSMs have been considered, where input sequences $\bar{x}_1, \bar{x}_2$ are said to *converge* in M if they reach the same state if applied to the initial state of M. I lift this definition to traces of OFSMs as follows:

Definition 2.5.1 (Convergence, divergence)**.** Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal OFSM and let $\alpha, \beta \in (\Sigma_I \times \Sigma_O)^*$ be two traces. Then α and β *converge* in M if they both reach the same state in M. That is, they converge if they satisfy the following condition

$$\alpha \in \mathcal{L}(M) \wedge \beta \in \mathcal{L}(M) \wedge M\text{-after-}\alpha = M\text{-after-}\beta$$

Analogously, α and β *diverge* in M if they reach distinct states:

$$\alpha \in \mathcal{L}(M) \wedge \beta \in \mathcal{L}(M) \wedge M\text{-after-}\alpha \neq M\text{-after-}\beta$$ ⊣

Note that it is not sufficient to define convergence of α and β simply as M-after-$\alpha = M$-after-β, since the *after*-operator is well defined only on traces in the language of an OFSM.

In the remainder of this section, I lift several other definitions and lemmata of [103, 106] to OFSMs. Convergence is discussed further in Section 4.2. The following lemma lifts Lemma 1 of [103] to OFSMs and introduces basic properties of convergence:

Lemma 2.5.2. *Suppose that α, β converge in OFSM M. Then the following holds for all traces γ over the alphabets of M:*

1. *If $\alpha.\gamma \in \mathcal{L}(M)$, then $\alpha.\gamma$ and $\beta.\gamma$ converge in M.*
2. *If $\gamma \in \mathcal{L}(M)$ does not converge with α in M, then γ also does not converge with β in M.* ⊣

Proof. By the assumption, α and β both reach some state q in M. Thus, if $\alpha.\gamma \in \mathcal{L}(M)$ holds, then

$$M\text{-after-}\alpha.\gamma = q\text{-after-}\gamma = M\text{-after-}\beta.\gamma$$

and hence (1.) follows. Analogously, if $\gamma \in \mathcal{L}(M)$ diverges from α, then (2.) follows from

$$M\text{-after-}\gamma \neq M\text{-after-}\alpha = M\text{-after-}\beta$$

by definition. □

In [103] and [106], test suites are generated that *preserve* the divergence or convergence of certain traces over FSMs that pass a test suite with respect to the reference model.

Definition 2.5.3 (Preservation of divergence and convergence)**.** Suppose $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ is a minimal OFSM and let A denote a set of OFSMs over input alphabet Σ_I and output alphabet Σ_O. Then a set $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ is *A-divergence-preserving* with respect to M, if for all $\alpha, \beta \in TS$ it holds that if α and β diverge in M, then α and β diverge in all elements of A. Analogously, TS is *A-convergence-preserving* with respect to M if for all $\alpha, \beta \in TS$ it holds that if α and β converge in M, then α and β converge in all elements of A. If a pair of traces α, β converges in all FSMs in A it is called *A-convergent*. ⊣

The preservation of divergence may be established via separating traces, as shown in the following lemma, which lifts Lemma 2 of [103] to OFSMs:

Lemma 2.5.4. *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal OFSM and let α, β diverge in M. Furthermore let A be a set of OFSMs over the same alphabets as M. Suppose that $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ is a test suite such that α and β are TS-separable. Finally suppose that for all $M' \in A$ it holds that $\mathcal{L}(M) \cap pref(TS) = \mathcal{L}(M') \cap pref(TS)$. Then α and β diverge in every element of A. That is, $\{\alpha, \beta\}$ is A-divergence-preserving.* ⊣

Proof. Following from the assumptions, there exists some trace ω separating α, β such that $\{\alpha.\omega, \beta.\omega\} \subseteq pref(TS)$. Since α, β diverge in M, it furthermore holds that $\alpha, \beta \in \mathcal{L}(M)$, implying

$$\omega \in \mathcal{L}_M(M\text{-after-}\alpha) \longleftrightarrow \omega \notin \mathcal{L}_M(M\text{-after-}\beta)$$

By $\mathcal{L}(M) \cap \mathit{pref}(TS) = \mathcal{L}(M') \cap \mathit{pref}(TS)$, the following also hold

$$\omega \in \mathcal{L}_M(M\text{-after-}\alpha) \longleftrightarrow \omega \in \mathcal{L}_{M'}(M'\text{-after-}\alpha)$$

$$\omega \in \mathcal{L}_M(M\text{-after-}\beta) \longleftrightarrow \omega \in \mathcal{L}_{M'}(M'\text{-after-}\beta)$$

from which it finally follows that ω also separates α, β in M':

$$\omega \in \mathcal{L}_{M'}(M'\text{-after-}\alpha) \longleftrightarrow \omega \notin \mathcal{L}_{M'}(M'\text{-after-}\beta)$$

This establishes that α, β diverge in M'. □

Conditions for the establishment of convergence are discussed further in Subsection 4.2.1 of Part II.

Chapter 3

Model-Based Testing with Finite State Machines

In the present work, I follow [114] in employing the term *model-based testing* (MBT) to describe the process of verifying a *system under test* (SUT) against a *reference model* specifying the allowed and expected behaviour, by generating from the latter test suites of certain guaranteed strengths, which are then systematically applied to the SUT via applying inputs and comparing observed responses with those exhibited by the reference model. In this process, so-called *complete* test suites are of high interest, as they guarantee fault coverage with respect to a given fault domain, which in the context of the present work are classes of FSMs. Fault coverage guarantees discussed here assume that the IO-behaviour of the SUT can be represented by an FSM within the fault domain.

A more detailed overview of model-based testing and its variants is presented in [114], while [2] contextualises MBT in the more general field of test case generation. MBT approaches have been developed for various formalisms more expressive than finite state machines, such as labelled transition systems as discussed in [111] or extended finite state machines as presented in [85].

In the following, I first discuss the conformance relations employed in the present work in Section 3.1. Next, Section 3.2 defines test suites and their completeness. Finally, Section 3.3 introduces the notion of *convergence graphs*, which are constructed concurrently with test suites in some test strategies and serve to store established convergences of certain traces in the test suite.

3.1 Conformance Relations

The most fundamental conformance relations employed for testing against finite state machines are *language-equivalence* and *reduction*. In the former case, an SUT *conforms* to reference model M if its IO-behaviour is equivalent to $\mathcal{L}(M)$. That is, M describes all allowed and expected behaviours. In contrast, in order for an SUT to conform to M with respect to reduction, it is sufficient for it to

exhibit no IO-trace not in $\mathcal{L}(M)$. That is, M models all allowed behaviours, but does not require the SUT to exhibit them all.

Definition 3.1.1 (Conformance)**.** Let M and I be FSMs over input alphabet Σ_I and output alphabet Σ_O. M and I are *language-equivalent*, denoted $I \sim M$, if their languages coincide:

$$I \sim M := \mathcal{L}(I) = \mathcal{L}(M)$$

Furthermore, I is a *reduction* of M, denoted $I \preceq M$, if all traces that are exhibited by I are also exhibited by M:

$$I \preceq M := \mathcal{L}(I) \subseteq \mathcal{L}(M) \qquad \dashv$$

The test strategies discussed in the present work generate test suites suitable for establishing language-equivalence. Parts of the corresponding mechanised formalisation have already been used successfully to implement strategies for reduction testing (see Appendix A).

From language-equivalence and reduction, several further conformance relations have been developed. For example, *quasi-equivalence* and *quasi-reduction* (examined in [43, 45, 87, 88, 89]) differ from the previous conformance relations by not constraining the behaviour of the SUT in cases where some state of the reference model has no emanating transition for some input x. Whereas language-equivalence and reduction require the corresponding states of the SUT to also not exhibit any response to x, their quasi-versions allow arbitrary responses to *undefined* inputs such as x. In [97], Peleska and I have introduced a further alternative conformance relation called *strong reduction*, which extends reduction by requiring that an input may be undefined in a conforming SUT if and only if it is undefined in the corresponding state of the reference model. Also note that many conformance relations on FSMs are similar to those developed for labelled transition systems (see, for example, [110]).

3.2 Complete Test Suites

In general, the language of an FSM is not finite. For example, in completely specified FSMs with at least one input symbol, each state must exhibit at least one emanating transition and hence paths of arbitrary length may be created. Thus, the behaviour of SUT or reference model cannot in general simply be exhaustively enumerated by a finite testing process. Therefore, the strategies discussed in the present work aim at generating finite test suites, containing only test cases of finite length, that are still sufficient to check whether the SUT conforms to the reference model.

In the following, I represent test cases as IO-traces and test suites as sets of test cases[1], using the following pass-relation to decide whether a SUT passes a test suite with respect to a reference model:

[1] Chapters 12 and 13 discuss alternative representations of test cases and test suites employed by the executable implementations of the test strategies discussed in Section 5.1. Appendix B justifies the use of IO-traces during the generation of test suites.

Definition 3.2.1 (Conformance on test suites)**.** Let M and I be FSMs over input alphabet Σ_I and output alphabet Σ_O. Furthermore let $A \subseteq (\Sigma_I \times \Sigma_O)^*$ be a set of IO-traces. Then M and I are *A-equivalent*, denoted $I \sim_A M$, if their languages coincide on intersection with A:

$$I \sim_A M := \mathcal{L}(I) \cap A = \mathcal{L}(M) \cap A$$

Furthermore, I is a *reduction* of M *on* A, denoted $I \preceq_A M$, if all traces in A that are exhibited by I are also exhibited by M:

$$I \preceq_A M := (\mathcal{L}(I) \cap A) \subseteq (\mathcal{L}(M) \cap A) \qquad \dashv$$

As the remainder of this work focusses on testing for language-equivalence, I write that an FSM I *passes* IO-trace α with respect to reference model M if M and I are $\{\alpha\}$-equivalent. Analogously, I passes test suite TS w.r.t. M if M and I are TS-equivalent.

In creating test suites, it is a common pattern to extend a set of traces in the language of the reference model with all traces up to a given length, potentially creating many traces with proper prefixes not contained in the language of the reference model. These may often be omitted by the following procedure:

Definition 3.2.2 (Shortening)**.** The *shortening* of a trace set $A \subseteq (\Sigma_I \times \Sigma_O)^*$ *for* FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$, denoted $short_M(A)$, is the trace set obtained by shortening all traces in A until all their proper prefixes are in the language of M:

$$short_M(A) = \{\alpha \mid \exists \alpha' \in A.\ \alpha \text{ is the longest prefix of } \alpha' \text{ such that } pref(\alpha) \setminus \alpha \subseteq \mathcal{L}(M)\} \qquad \dashv$$

The following lemma describes a case of testing for language-equivalence where it is sufficient to consider only shortened traces.

Lemma 3.2.3. *Let M and I be FSMs over input alphabet Σ_I and output alphabet Σ_O. Furthermore let $A \subseteq \mathcal{L}(M)$ be a non-empty set of traces. Let $B = \bigcup_{i=0}^{k} (\Sigma_I \times \Sigma_O)^i$ contain all traces up to length k for some $k \in \mathbb{N}$. Then the language-equivalence of M and I on A is preserved on the shortening of $A.B$:*

$$I \sim_{A.B} M \longleftrightarrow I \sim_{short_M(A.B)} M \qquad \dashv$$

Proof. Note first that $short_M(A.B)$ is the union of all $short_M(\{\alpha.\beta\})$ for $\alpha \in A, \beta \in B$, and hence that for $M' \in \{M, I\}$ it holds that

$$\mathcal{L}(M') \cap short_M(A.B) = \bigcup_{\alpha \in A, \beta \in B} (\mathcal{L}(M') \cap short_M(\{\alpha.\beta\}))$$

Since $A \subseteq \mathcal{L}(M)$ holds by assumption and B is prefix-closed, $short_M(A.B) \subseteq A.B$ follows and hence the forward implication holds.

For the converse implication assume that $\mathcal{L}(I) \cap short_M(A.B) \subseteq \mathcal{L}(M) \cap short_M(A.B)$ and suppose $I \sim_{A.B} M$ does not hold. Note first that $\alpha.\beta \in A.B \cap$

$\mathcal{L}(M)$ are not affected by shortening, as in this case $short_M(\{\alpha.\beta\}) = \{\alpha.\beta\}$ holds. Consider thus some $\alpha \in A, \beta \in B$ with $\alpha.\beta \notin \mathcal{L}(M)$ and $I \not\sim_{\{\alpha.\beta\}} M$, which in this case implies $\alpha.\beta \in \mathcal{L}(I)$. As $A \subseteq \mathcal{L}(M)$, there must exist traces β', β'' and an IO-pair x/y such that $\beta = \beta'.(x/y).\beta''$ and $short_M(\{\alpha.\beta\}) = \{\alpha.\beta'.(x/y)\}$. This implies $\alpha.\beta' \in \mathcal{L}(M)$ and also $\alpha.\beta'.(x/y) \notin \mathcal{L}(M)$. As languages of FSMs are prefix-closed, $\alpha.\beta'.(x/y) \in \mathcal{L}(I)$ follows from $\alpha.\beta \in \mathcal{L}(I)$. This contradicts the assumption of $\mathcal{L}(I) \cap short_M(A.B) \subseteq \mathcal{L}(M) \cap short_M(A.B)$. Hence, no $\alpha \in A, \beta \in B$ with $I \not\sim_{\{\alpha.\beta\}} M$ may exist, establishing $I \sim_{A.B} M$. □

By a similar argument, shortening also preserves the reduction conformance relation:

$$I \preceq_{A.B} M \longleftrightarrow I \preceq_{short_M(A.B)} M$$

3.2.1 Completeness

A test suite TS is *complete* for a given reference model M, conformance relation and fault domain $\mathcal{F}$, consisting of FSMs over the same alphabets as M, if each $I \in \mathcal{F}$ passes TS if and only if it conforms to M. This is usually split into *soundness* and *exhaustiveness* as follows (see [82, Section 2.5]). In the following I only consider completeness with respect to language-equivalence, allowing some simplification of the notation.

Definition 3.2.4 (Soundness, exhaustiveness, completeness)**.** Consider FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ and fault domain $\mathcal{F}$ where each $I \in \mathcal{F}$ uses input and output alphabets Σ_I and Σ_O, respectively. Finally, let $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ be some test suite. Then TS is *complete* with respect to M and $\mathcal{F}$ if both of the following conditions hold:

1. If $I \in \mathcal{F}$ is language-equivalent to M, then I passes M:

   $$I \sim M \longrightarrow I \sim_{TS} M$$

   This is called *soundness*[2].

2. If $I \in \mathcal{F}$ passes M, then I is language-equivalent to M.

   $$I \sim_{TS} M \longrightarrow I \sim M$$

   This is called *exhaustiveness*. ⊣

The strategies discussed and formalised in the present work (see Section 5.1) are parameterised by some $m \in \mathbb{N}$ and generate for a minimal observable reference model M test suites that are complete for the fault domain of all OFSMs over the same alphabets as M of at most m states. In the following, I denote this fault domain as $\mathcal{F}(M, m)$. Additionally, I consider only m such that $m \geq |M|$,

[2]Note that soundness holds trivially for language-equivalence and test suites represented as IO-traces, as I can only fail to pass TS w.r.t. M if there exists some trace in TS exhibited by only one of the FSMs.

implying $M \in \mathcal{F}(M, m)$. A test suite is called *m-complete* if it is complete for OFSM M and fault domain $\mathcal{F}(M, m)$. A test strategy is called m-complete if it generates m-complete test suites given m and a minimal OFSM M as reference model. Note that, while fault domains may contain infinitely many FSMs, the elements of $\mathcal{F}(M, m)$ may exhibit only finitely many distinct languages, since only the names of states are not bounded in $\mathcal{F}(M, m)$.

The following notation is employed to denote subsets of a fault domain that pass a test suite with respect to a given reference model.

Definition 3.2.5. Given an FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$, a set $\mathcal{F}$ of FSMs over input alphabet Σ_I and output alphabet Σ_O, and a test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$, I denote the set of all FSMs in $\mathcal{F}$ that are TS-equivalent to M as $\mathcal{F}_{TS}^M$. ⊣

In the following, I usually omit the superscript M in $\mathcal{F}_{TS}^M$ if reference model M is unambiguous from the context.

3.2.2 Assumptions on Systems Under Test

Complete test suites are of particular interest in the testing of safety-critical systems, as completeness provides a way to justify a certain test strength. In model-based testing, the system under test is usually not an FSM itself and hence certain assumptions are required to translate m-completeness of test suite to fault coverage capabilities against practical SUTs.

First, it is assumed that the behaviour (with respect to inputs and observed output responses) of the system under test can be represented as an FSM in the fault domain $\mathcal{F}(M, m)$ for reference model M. Parameter m thus places an upper bound on the complexity of considered behaviours and hence larger values of m increase the size of the fault domain and the number of SUTs satisfying the assumption – usually at the cost of increased test suite size.

Next, for nondeterministic SUTs, I apply the so-called *complete-testing assumption* (see [71]) that there exists some $k \in \mathbb{N}$ such that all responses of the SUT to any input sequence $\bar{x}$ are observed by applying $\bar{x}$ to the SUT at most k times. This assumption ensures that it can be decided in a finite number of steps whether some IO-trace $\bar{x}/\bar{y}$ is contained in the language of the SUT.

Finally, it is assumed that the SUT can be *reliably reset* to its initial state, for example by switching it off and then turning it on again[3].

Note that Hübner et al. have shown in [50] that test strategies based on complete test suites can exhibit significantly greater test strength than conventional random testing even on SUTs whose behaviour is outside of the fault domain. There also exist techniques such as [62, 93] for reducing the size of a test suite

[3] As the reset of an SUT often requires more time than applying some inputs to the SUT and observing the corresponding responses, the development of test strategies discussed in Chapter 5 has aimed at decreasing the number of required reset operations. Note that there exist complete strategies that do not assume the SUT to be resettable, generating test suites consisting of a single input sequence (see, for example, [112]). Such strategies instead assume that the reference model M is *strongly connected*, requiring that there exists a path between each pair of states of M.

after its construction while accepting the possible loss of completeness, thus proposing a trade-off between test suite size and fault coverage.

3.3 Convergence Graphs

During the generation of a test suite, only those convergences (see Section 2.5) can be exploited to minimise additions to the test suite, that have already been established to hold not only in the reference model, but also in the system under test. Thus, in addition to the iterative construction of test suites, strategies that exploit convergence are required to store information on which convergences can be safely used.

Definition 3.3.1 (Convergent class)**.** Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal OFSM and let $\alpha \in (\Sigma_I \times \Sigma_O)^*$ be some trace. I denote the set of all traces convergent with α in M as *convergent class* $[\alpha]^M$:

$$[\alpha]^M := \{\beta \mid \beta \text{ converges with } \alpha \text{ in } M\}$$

Furthermore, let $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ be a test suite and let $\mathcal{F}$ be some set of FSMs over Σ_I and Σ_O. Then let $[\alpha]_{TS}^{\mathcal{F}}$ denote the set of all traces in TS that converge with α in all FSMs in $\mathcal{F}$ which are TS-equivalent to M.

$$[\alpha]_{TS}^{M,\mathcal{F}} := \left\{\beta \in \mathit{pref}(TS) \;\middle|\; \forall I \in \mathcal{F}_{TS}^{M}.\ \beta \text{ converges with } \alpha \text{ in } I\right\} \qquad \dashv$$

If the reference model M and fault domain (usually $\mathcal{F}(M, m)$) are unambiguous in the context, I abbreviate $[\alpha]_{TS}^{M,\mathcal{F}(M,m)}$ to $[\alpha]_{TS}$.

In the following, I denote as *convergence graph* any data structure that is employed to store convergent classes during construction of a test suite TS for reference model M and supports the following operations:

- *Lookup*: The most important operation on a convergence graph G is the retrieval of the convergent class for a given trace α. In the following, this is denoted as function CG-LOOKUP(G, α), returning a set of traces (usually $[\alpha]_{TS}$ or a subset thereof).
- *Insertion*: Traces added to test suites during their computation are often also added into the convergence graph. In the following, this is supported by function CG-INSERT(G, α), returning a new convergence graph based on G (usually by inserting α into G).
- *Initialisation*: In all considered test strategies, an initial test suite TS is created without the use of a convergence graph. In the following, function CG-INITIALISE(TS) supports initialising a convergence graph using TS (usually by inserting all traces in TS into an empty graph).
- *Merging*: During the creation of test suites, it may be established that some α, β converge in M and all FSMs in the fault domain that pass the current test suite. As convergence is a transitive relation, this allows

unifying the convergent classes of α and β. In the following, function CG-MERGE(G, α, β) implements this merging operation.

Implementations of convergence graphs are discussed in [103, 106, 108].

Implementations and usages of convergence graphs are not required to return all traces in $[\alpha]_{TS}$ for CG-LOOKUP(G, α). For the correctness of the strategies exploiting convergence, it is only required that CG-LOOKUP(G, α) does not return traces not converging with α w.r.t. test suite TS. For convenience, CG-LOOKUP(G, α) should also return α itself. I employ the notion of *validity* of convergence graphs to express these requirements in subsequent chapters. That is, a convergence graph is *valid* if for each $\alpha \in \mathcal{L}(M)$ it holds that $\{\alpha\} \subseteq$ CG-LOOKUP$(G, \alpha) \subseteq [\alpha]_{TS}$. Prior to Chapter 9 of Part III, which formalises implementations of convergence graphs, I assume that the insertion, initialisation, and merge operations preserve validity under the following assumptions:

- CG-INITIALISE(TS) returns a valid convergence graph.
- CG-INSERT(G, α) returns a valid convergence graph if G is valid and $\alpha \in \mathcal{L}(M)$ holds.
- CG-MERGE(G, α, β) returns a valid convergence graph if G is valid and α, β converge in M.

The most common operation using convergence graphs employed by test strategies discussed in this work is to select for some given traces α, γ some $\beta \in [\alpha]_{TS}$ after which to append γ, inserting the resulting trace into the test suite. This is implemented as DISTRIBUTEEXTENSION in Algorithm 1. Note that this algorithm does not specify how β is selected. Function DISTRIBUTEEXTENSION additionally inserts the longest prefix of $\beta.\gamma$ in $\mathcal{L}(M)$ into the convergence graph.

Algorithm 1: DISTRIBUTEEXTENSION$(M, \alpha, \gamma, TS, G)$

```
1 choose some β ∈ CG-LOOKUP(G,α)
2 TS ← β.γ
3 ω ← the longest prefix of α.γ in L(M)
4 CG-INSERT(G, ω)
```

Part II

Equivalence-Testing with Finite State Machines

Chapter 4

Sufficient Conditions for Completeness

In the development of test strategies based on finite state machines, several authors have proven m-completeness of their proposed strategies by first establishing sufficient (but not always necessary) conditions for m-completeness of test suites, and then showing that their strategies satisfy those conditions. Such conditions include the H-Condition [27], the SPY-Condition [102, 103] and the S-Condition [106]. In addition to these conditions developed in order to establish completeness of certain strategies, other studies such as [12, 13, 14, 100, 112, 116] have investigated sufficient and sometimes also necessary conditions for m-completeness in order to develop algorithms that check whether a given test suite is m-complete for a given reference model.

In this chapter, I describe sufficient conditions for m-completeness that are suited to guide the generation of test suites in that they exhibit sub-conditions that can be established iteratively during construction of a test suite. For example, the H-Condition requires certain pairs of traces to be extended with distinguishing traces, which can be implemented and also verified in isolation for each pair. I group the considered sufficient conditions into those conditions that describe which test cases need to be extended with distinguishing traces, thus ensuring divergences, and those conditions that describe which traces need to be established as converging in the system under test. Section 4.1 introduces the former, while Section 4.2 introduces the latter.

4.1 Completeness from Divergence

4.1.1 H-Condition

The H-Condition has first been proposed in [27] and constitutes a sufficient condition for m-completeness of test suites for language-equivalence testing. It can be split into two conditions: First, a state cover of the reference model

should be extended with test cases of length up to $m - n + 1$, where n is the size of the reference model[1]. Second, certain pairs of test cases in the resulting test case set should be further extended with distinguishing test cases. The H-Condition has originally been proposed to describe test suites consisting of input sequences as described in the following Lemma.

Lemma 4.1.1 (Input-based H-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ and $I = (T, t_0, \Sigma_I, \Sigma_O, h_T)$ be minimal observable FSMs with $|Q| = n$ and $|T| \leq m$ such that $m \geq n$. Furthermore let V be a state cover of M and construct sets A, B, C as follows, where $V|_{\Sigma_I}$ denotes the projection of V to input sequences:*

$$A := V|_{\Sigma_I} \times V|_{\Sigma_I}$$

$$B := V|_{\Sigma_I} \times (V|_{\Sigma_I}. \bigcup_{i=1}^{m-n+1} \Sigma_I^i)$$

$$C := \left\{ (\bar{v}\bar{x}', \bar{v}\bar{x}) \;\middle|\; \bar{v} \in V|_{\Sigma_I} \wedge \bar{x} \in \bigcup_{i=1}^{m-n+1} \Sigma_I^i \wedge \bar{x}' \in pref(\bar{x}) \setminus \{\bar{x}\} \right\}$$

Then any test suite $TS \subseteq \Sigma_I^*$ such that

1. *$V.\bigcup_{i=0}^{m-n+1} \Sigma_I^i \subseteq TS$, and*
2. *for all $(\alpha, \beta) \in A \cup B \cup C$ such that α and β reach distinct states q, q' in M, there exists some input sequence ω distinguishing q, q' such that $\alpha.\omega, \beta.\omega \in TS$,*

is m-complete. That is, the following holds:

$$\mathcal{L}(I) = \mathcal{L}(M) \longleftrightarrow \{\bar{x}/\bar{y} \in \mathcal{L}(I) \mid \bar{x} \in TS\} = \{\bar{x}/\bar{y} \in \mathcal{L}(M) \mid \bar{x} \in TS\} \qquad \dashv$$

This result can be generalised to a language-theoretic insight, as described and proven in [82, Section 4.7], resulting in the following Lemma:

Lemma 4.1.2 (Language-based H-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ and $I = (T, t_0, \Sigma_I, \Sigma_O, h_T)$ be minimal observable FSMs with $|Q| = n$ and $|T| \leq m$ such that $m \geq n$. Furthermore let V be a state cover of M and let*

$$A := V \times V$$

$$B := V \times (V. \bigcup_{i=1}^{m-n+1} (\Sigma_I \times \Sigma_O)^i)$$

$$C := \left\{ (\upsilon.\alpha, \upsilon.\beta) \;\middle|\; \upsilon \in V \wedge \beta \in \bigcup_{i=1}^{m-n+1} (\Sigma_I \times \Sigma_O)^i \wedge \alpha \in pref(\beta) \setminus \{\beta\} \right\}$$

Then any test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ such that

[1] As it is assumed here that $m \geq n$ holds, this extension of a state cover contains a transition cover of the reference model.

1. $V.\bigcup_{i=0}^{m-n+1}(\Sigma_I \times \Sigma_O)^i \subseteq TS$, *and*
2. *for all* $(\alpha, \beta) \in A \cup B \cup C$ *such that* M*-after-*$\alpha \neq M$*-after-*β*, there exists some trace* $\omega \in \Delta_M(M\text{-after-}\alpha, M\text{-after-}\beta)$ *such that* $\alpha.\omega, \beta.\omega \in TS$

is m*-complete for* M*. That is, the following holds:*

$$\mathcal{L}(I) = \mathcal{L}(M) \longleftrightarrow I \sim_{TS} M \qquad \dashv$$

Note that Lemma 4.1.1 is a direct result of Lemma 4.1.2, as any test suite $TS \subseteq \Sigma_I^*$ satisfying the conditions of Lemma 4.1.1 can be transformed into a test suite satisfying the conditions of Lemma 4.1.2 by replacing each $\bar{x} \in TS$ with all $\bar{x}/\bar{y}$ such that $\bar{y} \in \Sigma_O^{|\bar{x}|}$.

The conditions used in Lemma 4.1.2 can be further weakened in two ways: First, via Lemma 3.2.3 it is not necessary to consider extensions β of traces $v.\alpha \notin \mathcal{L}(M)$ where $v \in V$. Thus, in condition (1.) it is sufficient to extend each trace reaching some $q \in Q$ only with traces whose proper prefixes are contained in $\mathcal{L}_M(q)$. Second, condition (2.) of Lemma 4.1.2 is not limited to traces that do reach states in the reference model and hence may require appending distinguishing traces after traces not in $\mathcal{L}(M)$. By considering only pairs of traces in $\mathcal{L}(M)$, such cases may be avoided.

The following lemma introduces these modifications, introducing a set X_q for each $q \in Q$ to denote the set of all non-empty traces of length up to $m - n + 1$ over $\Sigma_I \times \Sigma_O$ whose proper prefixes are all contained in $\mathcal{L}_M(q)$. The result constitutes a further generalisation of Lemma 4.1.2.

Lemma 4.1.3 (Weakened H-Condition)**.** *Let* $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ *and* $I = (T, t_0, \Sigma_I, \Sigma_O, h_T)$ *be minimal observable FSMs with* $|Q| = n$ *and* $|T| \leq m$ *such that* $m \geq n$*. Furthermore let* V *be a state cover of* M *and for each* $q \in Q$ *let* X_q *be defined as follows:*

$$X_q := \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$$

Finally let

$$\begin{aligned} A &:= V \times V \\ B &:= V \times \{v_q.\alpha \mid q \in Q \wedge \alpha \in X_q\} \\ C &:= \{(v_q.\alpha, v_q.\beta) \mid q \in Q \wedge \beta \in X_q \wedge \alpha \in pref(\beta) \setminus \{\beta\}\} \end{aligned}$$

Then any test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ *satisfying the following conditions is* m*-complete for* M*:*

1. $V \cup \{v_q.\alpha \mid q \in Q \wedge \alpha \in X_q\} \subseteq TS$, *and*
2. *for all* $(\alpha, \beta) \in A \cup B \cup C$ *such that* $\alpha, \beta \in \mathcal{L}(M)$ *and* M*-after-*$\alpha \neq M$*-after-*β*, there exists some trace* $\omega \in \Delta_M(M\text{-after-}\alpha, M\text{-after-}\beta)$ *such that* $\alpha.\omega, \beta.\omega \in TS$*.* $\dashv$

I omit a detailed proof, as this result follows from combining Lemma 4.1.2 with Lemma 3.2.3. Fig. 5.5 of Subsection 5.1.5 visualises how property (2.) is satisfied via the H-Method.

4.1.2 S-Condition

When considering some pair (α, β) in property (2.) of Lemma 4.1.3, the H-Condition allows arbitrary choices of separating traces γ, but explicitly requires appending them after α and β in order to check whether γ is contained in the languages of the states reached by α, β. As discussed in Section 2.5, the same information may be gained by appending γ to traces α', β' that respectively converge with α and β. The S-Condition developed by Soucha in [106] generalises the H-Condition on completely specified DFSMs (see Lemma 4.1.1) by allowing this additional choice of selecting convergent traces after which to append separating traces, which enables more and potentially smaller test suites to satisfy this sufficient condition for completeness.

In order to replace $\alpha.\gamma$ with $\alpha'.\gamma$ as described above while retaining completeness, it is first required to establish convergence of α and α' in both the reference model M and the system under test I, since otherwise $\alpha.\gamma$ and $\alpha'.\gamma$ may apply γ to distinct states of M or I. Conditions for establishing convergences in the SUT are discussed in Subsection 4.2.1. Recall notation $\mathcal{F}(M,m)_{TS}$ denoting the set of all $I \in \mathcal{F}(M,m)$ that pass TS with respect to M and notation $[\alpha]_{TS}$ denoting the set of all α' converging with α in all $I \in \mathcal{F}(M,m)_{TS}$. Also recall that $\Delta(q,q')$ denotes the set of all traces distinguishing q and q'. Let $\Delta(q,q')|_{\Sigma_I}$ denote the input projection of $\Delta(q,q')$. Then the following lemma presents the input-based formulation of the S-Condition as presented by Soucha in Theorem 8.14 of [106].

Lemma 4.1.4 (S-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal completely specified DFSM with $|Q| = n$. Furthermore let V be a state cover of M and let $n \leq m$. Then any test suite $TS \subseteq \Sigma_I^*$ satisfying the following conditions is complete for M and the set of all completely specified DFSMs in $\mathcal{F}(M,m)$:*

1. *For each $q \in Q$ and sequence $\bar{x}$ over Σ_I of length up to $m-n+1$, $\bar{x}$ is appended to some sequence convergent with $\bar{v}_q$:*

$$\forall q \in Q, \bar{x} \in \Sigma_I^{\leq m-n+1} \; \exists \bar{u} \in [\bar{v}_q]_{TS}.\, \bar{u}.\bar{x} \in TS$$

2. *For all $q, q' \in Q$ and sequences $\bar{x}$ over Σ_I of length up to $m-n+1$ such that $\bar{v}_q.\bar{x}$ and $\bar{v}_{q'}$ diverge in M, TS establishes that $\bar{v}_q.\bar{x}$ and $\bar{v}_{q'}$ diverge in $\mathcal{F}(M,m)_{TS}$, for example by adding distinguishing sequences, which can be distributed over convergent sequences:*

$$\begin{aligned}&\forall q, q' \in Q, \bar{x} \in \Sigma_I^{\leq m-n+1}.\, q\text{-after-}\bar{x} \neq q' \longrightarrow \\ &\quad \exists \bar{u} \in [\bar{v}_q]_{TS}, \bar{u}' \in [\bar{v}_{q'}]_{TS}, \bar{w} \in \Delta(q\text{-after-}\bar{x}, q')|_{\Sigma_I}. \\ &\qquad \{\bar{u}.\bar{x}.\bar{w}, \bar{u}'.\bar{w}\} \subseteq TS\end{aligned}$$

3. *For all $q \in Q$, sequences $\bar{x}$ over Σ_I of length up to $m-n+1$, and proper prefixes $\bar{x}'$ of $\bar{x}$ such that $\bar{v}_q.\bar{x}'$ and $\bar{v}_q.\bar{x}$ diverge in M, TS establishes that $\bar{v}_q.\bar{x}'$ and $\bar{v}_q.\bar{x}$ diverge in $\mathcal{F}(M,m)_{TS}$, for example by adding distinguishing*

sequences, which can be distributed over convergent sequences:

$$\forall q, q' \in Q, \bar{x} \in \Sigma_I^{\leq m-n+1}, \bar{x}' \in \mathit{pref}(\bar{x}) \setminus \{\bar{x}\}.\ q\text{-}\mathit{after}\text{-}\bar{x}' \neq q\text{-}\mathit{after}\text{-}\bar{x} \longrightarrow$$
$$\exists \bar{u}, \bar{u}' \in [\bar{v}_q]_{TS}, \bar{w} \in \Delta(q\text{-}\mathit{after}\text{-}\bar{x}', q\text{-}\mathit{after}\text{-}\bar{x})|_{\Sigma_I}.$$
$$\{\bar{u}'.\bar{x}'.\bar{w}, \bar{u}.\bar{x}.\bar{w}\} \subseteq TS \qquad \dashv$$

Thus, Lemma 4.1.4 generalises Lemma 4.1.1 by introducing the option of distribution over convergent sequences to conditions (1.) and (2.) of Lemma 4.1.1. The generalisation of the latter condition is split into conditions (2.) and (3.) of Lemma 4.1.4 to simplify the presentation, as sets of pairs A, B, and C used in Lemma 4.1.1 do not describe how contained sequences are to be split into a prefix obtained from the state cover and a suffix representing an extension of length up to $m - n + 1$.

Note here that the S-Condition does not prescribe which input sequences are to be established as convergent, how convergence is to be established, or in which order the conditions (1.) to (3.) or aspects thereof are to be satisfied during test suite construction. One possible implementation can be found in the SPYH-Method (see [106]).

The S-Condition can be generalised to arbitrary OFSMs analogously to the generalisation of the input-based H-Condition (Lemma 4.1.1) to the IO-based H-Condition of Lemma 4.1.3. That is, it can be generalised by first converting all references to input sequences into references to IO-traces, followed by constricting the conditions such that condition (1.) only considers traces whose proper prefixes are in the language of q, while conditions (2.) and (3.) only consider extensions that are in the language of q. The following lemma presents the result of this generalisation.

Lemma 4.1.5 (Generalised S-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal observable FSM with $|Q| = n$. Furthermore let V be a state cover of M, $n \leq m$, and for each $q \in Q$ let X_q be defined as follows*

$$X_q := \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$$

Then any test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ satisfying the following conditions is m-complete over $\mathcal{F}(M, m)$:

1. *For each $q \in Q$ and trace $\alpha \in X_q$, α is appended to some trace convergent with v_q:*
$$\forall q \in Q, \alpha \in X_q.\ \exists \omega \in [v_q]_{TS}.\ \omega.\alpha \in TS$$

2. *For all $q, q' \in Q$ and traces $\alpha \in X_q \cap \mathcal{L}_M(q)$, if $v_q.\alpha$ and $v_{q'}$ diverge in M, then TS establishes that $v_q.\alpha$ and $v_{q'}$ diverge in $\mathcal{F}(M, m)_{TS}$, for example by adding distinguishing traces, which can be distributed over convergent traces:*
$$\forall q, q' \in Q, \alpha \in X_q \cap \mathcal{L}_M(q).\ q\text{-}\mathit{after}\text{-}\alpha \neq q' \longrightarrow$$
$$\exists \omega \in [v_q]_{TS}, \omega' \in [v_{q'}]_{TS}, \gamma \in \Delta(q\text{-}\mathit{after}\text{-}\alpha, q').\ \{\omega.\alpha.\gamma, \omega'.\gamma\} \subseteq TS$$

3. *For all $q \in Q$, traces $\alpha \in X_q \cap \mathcal{L}_M(q)$, and proper prefixes α' of α such that $v_q.\alpha'$ and $v_q.\alpha$ diverge in M, TS establishes that $v_q.\alpha'$ and $v_q.\alpha$ diverge in $\mathcal{F}(M,m)_{TS}$, for example by adding distinguishing traces, which can be distributed over convergent traces:*

$$\forall q \in Q, \alpha \in X_q \cap \mathcal{L}_M(q), \alpha' \in pref(\alpha) \setminus \{\alpha\}.\ q\text{-}after\text{-}\alpha' \neq q\text{-}after\text{-}\alpha$$
$$\exists \omega, \omega' \in [v_q]_{TS}, \gamma \in \Delta(q\text{-}after\text{-}\alpha', q\text{-}after\text{-}\alpha).\ \{\omega'.\alpha'.\gamma, \omega.\alpha.\gamma\} \subseteq TS \ \dashv$$

This lemma follows from Lemma 4.1.4 and Lemma 3.2.3. I omit a detailed proof here, as the next subsection introduces a more abstract sufficient condition for completeness in Lemma 4.1.7, which is satisfied by any test suite that satisfies the conditions given in the above Lemma 4.1.5. Fig. 5.7 of Subsection 5.1.7 visualises how the properties (2.) and (3.) are satisfied by the SPYH-Method.

4.1.3 Indirect Sufficiency

Conversely to deriving the H-Condition from the S-Condition, it is also possible to derive sufficiency of the S-Condition from sufficiency of the H-Condition. Suppose that $\beta \in [\alpha]_{TS}$ and $\beta.\gamma \in TS$ hold. Then for any $I \in \mathcal{F}(M,m)_{TS}$ it holds by Lemma 2.5.2 that

$$\beta.\gamma \in \mathcal{L}(I) \longleftrightarrow \alpha.\gamma \in \mathcal{L}(I)$$

and hence that I also passes a test suite in which $\beta.\gamma$ is replaced by $\alpha.\gamma$. Therefore, if TS is a test suite satisfying the S-Condition for M and m, then a test suite TS' satisfying the H-Condition can be derived from TS by replacing every choice of some $\beta \in [\alpha]_{TS}$ with α itself. Any I passing TS thus also passes TS'. As TS' is m-complete, so is TS. The following lemma generalises this property, which follows immediately from the definition of m-completeness:

Lemma 4.1.6 (Indirect Sufficiency)**.** *Let TS and TS' be test suites and suppose that TS' is complete for $\mathcal{F}(M,m)$ and OFSM M. Suppose that any $I \in \mathcal{F}(M,m)$ that passes TS also passes TS', that is*

$$\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_{TS'}$$

Then TS is m-complete for reference model M. $\dashv$

Furthermore, it is possible to define a more abstract sufficiency condition than the H and S-Conditions by removing explicit requirements on how preservation of divergence is to be established. This approach, formalised in the following lemma, extracts the core property of test suites satisfying the H or S-Condition – the fact that they preserve divergence between state cover traces and along extensions appended to the state cover – without concrete conditions on the way this property is established. I omit a proof at this point, as a mechanised proof of the lemma in Isabelle/HOL is discussed in Subsection 10.1.1 and visualised in Fig. 10.1.

Lemma 4.1.7 (Abstract H-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal observable FSM with $|Q| = n$ and let $I \in \mathcal{F}(M, m)$. Furthermore let V be a state cover of M, $n \leq m$, and for each $q \in Q$ let X_q be defined as follows*

$$X_q := \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$$

Suppose that for each $q \in Q$ and $\gamma \in X_q$ the following holds

1. *I passes $V \cup \{v_q\}.pref(\gamma)$, that is*

$$\mathcal{L}(I) \cap (V \cup \{v_q\}.pref(\gamma)) = \mathcal{L}(M) \cap (V \cup \{v_q\}.pref(\gamma))$$

2. *$V \cup \{v_q\}.pref(\gamma)$ is $\{I\}$-divergence-preserving. That is, traces $\alpha, \beta \in V \cup \{v_q\}.pref(\gamma)$ that diverge in M also diverge in I.*

Then, I and M are language-equivalent. ⊣

Note that Lemma 4.1.7 does not specify which test cases a test suite should contain. Instead, it provides an intermediate property such that any test suite sufficient to establish this property is complete. In my formalisation in Isabelle/HOL (see Section 10.1), I omit explicit usage of the S-Condition and instead employ the more abstract conditions of Lemma 4.1.7.

4.2 Completeness from Convergence

The H and S-Conditions describe that a test suite is m-complete if it contains a certain set of traces such that certain pairs of traces are furthermore extended by shared distinguishing traces. Thus, they establish completeness by describing which traces in the test suite should lead to distinct states in all FSMs in $\mathcal{F}(M, m)$ that pass the test suite. In contrast, the SPY-Condition, introduced in [102, 103][2], shows that a test suite TS is complete if it establishes convergence of certain traces that form a transition cover of the reference model.

The following lemma generalises the SPY-Condition, originally stated in Theorem 1 of [102] only for completely specified DFSMs and using input sequences, to partial nondeterministic OFSM reference models and IO-traces. In contrast to the presentation of the H and S-Conditions, I here omit the original formulation, as it is identical with the formulation of the later Corollary 4.2.2 except for employing the concepts of transition cover and convergence with respect to input sequences instead of IO-traces.

Lemma 4.2.1 (Generalised SPY-Condition)**.** *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal observable FSM. Then a test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ is complete for fault domain $\mathcal{F}$ if and only if it contains an $\mathcal{F}_{TS}$-convergence-preserving transition cover A such that $\epsilon \in A$.* ⊣

[2] The name *SPY-Condition* has been introduced later in [108].

A proof for of this lemma for completely specified DFSMs is presented in [102, Theorem 1], which may easily be extended to minimal OFSMs by suitable liftings of the concepts of transition covers and convergence to nondeterministic FSMs, as presented in Sections 2.3 and 2.5, respectively.

By using $\mathcal{F}(M, m)$ as fault domain, this result can then be used to explicitly describe a sufficient condition for m-completeness:

Corollary 4.2.2. *Consider minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$. Then a test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ is m-complete if it contains an $\mathcal{F}(M, m)_{TS}$-convergence-preserving transition cover A such that $\epsilon \in A$.* ⊣

I present a mechanised formalisation of this result in Section 9.1.

4.2.1 Sufficient Conditions for Establishing Convergence

In [102, 103], Simão et al. also provide a sufficient condition to decide whether a test suite preserves convergence of certain traces, which is not given in the SPY-Condition itself. This condition is proven correct using an argument over the assumed upper bound m on the states of the system under test, relying on two auxiliary lemmata. I discuss these here in detail, as one of the steps in [102, 103] has been shown to contain a flaw.

First, a lower bound is established on the number of states of a system under test in the case that a pair of traces convergence in the reference model but not in the SUT, where the traces are part of a test suite providing certain further information on the behaviour of these traces in any FSM passing it.

Lemma 4.2.3. *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ and $I = (T, t_0, \Sigma_I, \Sigma_O, h_I)$ be minimal OFSMs. Suppose that $\alpha, \beta \in \mathcal{L}(M) \cap \mathcal{L}(I)$ converge in M but not in I. Furthermore suppose that $\omega \in \Delta_I(I\text{-after-}\alpha, I\text{-after-}\beta)$ is a minimal length trace distinguishing the states reached by α, β in I. Let ω' be some proper prefix of ω such that $\omega \in \mathcal{L}_M(M\text{-after-}\alpha) \cap \mathcal{L}_M(M\text{-after-}\beta)$ and suppose that $\{\alpha, \beta\}.pref(\omega')$ is $\{I\}$-divergence-preserving. Finally suppose that I passes $\{\alpha, \beta\}.pref(\omega')$ with respect to M, that is, that the following holds:*

$$\mathcal{L}(M) \cap \{\alpha, \beta\}.pref(\omega') = \mathcal{L}(I) \cap \{\alpha, \beta\}.pref(\omega')$$

Then the traces in $\{\alpha, \beta\}.pref(\omega')$ reach at least

$$|\omega'| + \left| \bigcup_{\omega'' \in pref(\omega')} M\text{-after-}\alpha.\omega'' \right| + 1$$

distinct states of I. ⊣

I once more omit a detailed proof at this point, as the lemma is a generalisation to partial nondeterministic OFSMs of Lemma 8.7 of [106], which is in turn a modification of Lemma 3 of [103]. This latter modification has become necessary, as the lemma in [103] did not use strong enough assumptions to

prove the claimed result[3]. This flaw has been first amended by Michal Soucha in Lemma 8.7 of [106]. Wen-ling Huang and I also independently observed this flaw, whereupon Wen-ling Huang developed another amended proof of Theorem 2 of [103] – which employs Lemma 3 – making use of the stronger assumptions of this theorem. A mechanised proof of it is part of my formalisation using Isabelle/HOL described in Part III[4]. A proof of Lemma 4.2.3 as stated above is discussed in Subsection 9.1.1.

The following general property on the maximum number of states visited by a minimal distinguishing trace constitutes the second required lemma:

Lemma 4.2.4. *Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ be a minimal OFSM and let $q, q' \in Q$. Suppose that ω is a minimal length distinguishing trace for q and q'. Finally let $S \subseteq Q$ be some non-empty set of states of M. Then the following holds:*

$$\left|\left\{\omega' \mid \omega' \in pref(\omega) \setminus \{\omega\} \wedge q\text{-}after\text{-}\omega' \in S \wedge q'\text{-}after\text{-}\omega' \in S\right\}\right| \leq |S| - 1$$

That is, the number of proper prefixes ω' of ω that reach states in S from both q and q' is at most $|S| - 1$. ⊣

The above lemma is a generalisation of Lemma 8.8 of [106] and its accompanying proof to IO-traces. A proof of the mechanised formalisation of this lemma is discussed in Subsection 8.2.5.

Combining this result with Lemma 4.2.3, it is now possible to prove a sufficient condition on convergence, which is based on extending the traces to be shown to converge up to length $m - n$ while preserving divergence. This sufficient condition is a generalisation of Theorem 2 of [103] (Lemma 8.6 in [106]) to partial nondeterministic OFSMs.

Lemma 4.2.5. *Let $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ be a test suite for minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ where $|Q| = n$. Let $n \leq m$. Suppose that $\pi, \tau \in \mathcal{L}(M)$ converge in state q of M. Furthermore suppose that for each $\gamma \in (\Sigma_I \times \Sigma_O)^*$ such that $|\gamma| \leq m - n$ and each proper prefix of γ is contained in $\mathcal{L}_M(q)$, there exist some $\alpha \in [\pi]_{TS}$, $\beta \in [\tau]_{TS}$ and a state cover $SC \subseteq TS$ of M that contains $\{\alpha, \beta\}.pref(\gamma)$ and is $\mathcal{F}(M, m)_{TS}$-divergence-preserving. Then π and τ are $\mathcal{F}(M, m)_{TS}$-convergent.* ⊣

[3] More precisely, Lemma 3 of [103] does not assume ω to be a minimal length trace separating α, β in I, but instead merely assumes that ω is a shortest trace such that I does not pass $\{\alpha.\omega, \beta.\omega\}$ with respect to M. This assumption does not imply that ω separates α, β in I, as ω may now simply be a trace distinguishing both states reached by α, β in I from M-after-$\alpha = M$-after-β. Consider, for example, the case where I-after-α and I-after-β both exhibit a transition labelled $(a/1)$ to the same state t, which in turn satisfies $a/2 \in \mathcal{L}_I(t)$, whereas it holds that $a/1 \in \mathcal{L}_M(M\text{-after-}\alpha)$ but $aa/12 \notin \mathcal{L}_M(M\text{-after-}\alpha)$. Additionally suppose that applying $a/1$ to M-after-α reaches a state q other than M-after-α and that $\omega = aa/12$ is a shortest trace such that I fails to pass $\{\alpha.\omega, \beta.\omega\}$. Then, due to the convergence of $\alpha.(a/1)$ and $\beta.(a/1)$ in I, for proper prefix $\omega' = a/1$ of ω the traces in $\{\alpha, \beta\}.pref(\omega')$ reach only 3 distinct states of I. This is less than the lower bound of $|\omega'| + |\{M\text{-after-}\alpha, q\}| + 1 = 4$ distinct states claimed by the lemma. A more detailed discussion of this flaw in Lemma 3 of [103] can be found in Appendix D.2 of [106].

[4] See lemma `sufficient_condition_for_convergence_in_SPY_method` available in theory file `Convergence.thy`, discussed in Section 9.1

Compared to the original condition from [103], the above Lemma incorporates a slight optimisation for partial nondeterministic OFSMs by only considering traces whose proper prefixes are in the language, thus avoiding application of superfluous traces. I omit a proof here, as this lemma follows from Lemma 4.2.6, as developed below. A mechanised formalisation and proof of these two lemmata is discussed in Subsection 9.1.2 of Part III.

Lemma 4.2.5 enables the iterative creation of an m-complete test suite TS from a state cover V by considering in turn each transition (q, x, y, q') of the reference model and establishing the $\mathcal{F}(M,m)_{TS}$-convergence of $\upsilon_q.(x/y)$ and $\upsilon_{q'}$ by adding suitable test cases. After also adding $\upsilon_q.(x/y)$ for all states q and $x/y \notin \mathcal{L}_M(q)$ to TS, this constitutes a convergence preserving transition cover as required by the SPY-Condition (Corollary 4.2.2).

In practical testing, Lemma 4.2.5 is not always applied to prove π and τ to be $\mathcal{F}(M,m)_{TS}$-convergent by ensuring that test suite TS contains the exact sets $\{\alpha, \beta\}.pref(\gamma)$ required in the lemma. Instead, analogous to Lemma 4.1.6, it is ensured that there exists some test suite TS' containing these exact sets for a fixed state cover of M such that $\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_{TS'}$. Based on this approach, the following lemma can be derived, which simplifies some proofs:

Lemma 4.2.6. *Let $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ be a test suite for minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ where $|Q| = n$. Let $n \leq m$. Suppose that $\pi, \tau \in \mathcal{L}(M)$ converge in state q of M. Furthermore suppose that there exists some test suite TS' such that $\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_{TS'}$ and a state cover V of M satisfying the condition that for all $\gamma \in \bigcup_{i=0}^{m-n}(\Sigma_I \times \Sigma_O)^i$ such that $pref(\gamma) \setminus \{\gamma\} \subseteq \mathcal{L}_M(q)$, both*

1. *$V \cup \{\pi, \tau\}.pref(\gamma) \subseteq TS'$ holds, and*
2. *$V \cup \{\pi, \tau\}.pref(\gamma)$ is $\mathcal{F}(M,m)_{TS}$-divergence-preserving.*

Then π and τ are $\mathcal{F}(M,m)_{TS}$-convergent. ⊣

Note that the use of Lemma 4.2.6 to establish convergences in order to construct the convergence-preserving transition cover presupposed by the SPY-Condition (Corollary 4.2.2) requires establishing preservation of divergence similar to that required by the generalised S-Condition (Lemma 4.1.5). To my best knowledge, Lemma 4.2.6 provides the only sufficient condition for convergence that is applicable to arbitrary minimal observable reference models M and fault domain $\mathcal{F}(M,m)$, and hence current test strategies that generate test suites satisfying the SPY-Condition also satisfy the generalised S-Condition.

Chapter 5

Overview of Complete Test Strategies

This chapter provides an overview of test strategies that are complete for testing language-equivalence on FSMs. Section 5.1 details the test strategies formalised in this work, while Section 5.2 lists further strategies that are outside of the scope of this work for various reasons.

Throughout this chapter, I employ minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$ with $|M| = n$ as the reference model, against which m-complete test suites for fault domain $\mathcal{F}(M, m)$ are to be computed, where $m \geq n$ is the assumed upper bound on the number of states of the minimal OFSM representing the behaviour of the system under test.

5.1 Selected Strategies

In this work, I have formalised most test strategies that generate m-complete test suites for arbitrary $m \geq n$. These include the W, Wp, HSI, H, SPY, and SPYH-Methods as introduced in subsequent subsections. I have not considered strategies that are complete only for proper subsets of fault domain $\mathcal{F}(M, m)$, as well as strategies that place further requirements on the reference model, such as the existence of unique distinguishing sequences (see Section 5.2). This exclusion follows from the goal of developing generic frameworks with which to implement and proof complete the test strategies, which is greatly complicated by considering strategies with conflicting assumptions on the reference model or desired completeness properties.

Most complete strategies for testing w.r.t. language-equivalence have originally been proposed for deterministic and completely specified FSMs. Thus, the first part of each strategy description provided in this chapter describes how an m-complete test suite using input sequences as test cases is computed. Thereafter, I present a simple pseudocode implementation of the strategy generalised to partial and nondeterministic reference models, using IO-traces as test cases.

Completeness proofs and concrete implementations are discussed in Part III of this work, which describes the formalisation of generalised implementations of these strategies in Isabelle/HOL.

5.1.1 Running Example

Throughout this section, I employ OFSM M_5 depicted in Fig. 5.1 as a running example in using the various test strategies to compute 4-complete test suites. The input and output alphabets of M_5 are $\Sigma_I = \{a, b, c\}$ and $\Sigma_O = \{0, 1\}$, respectively.

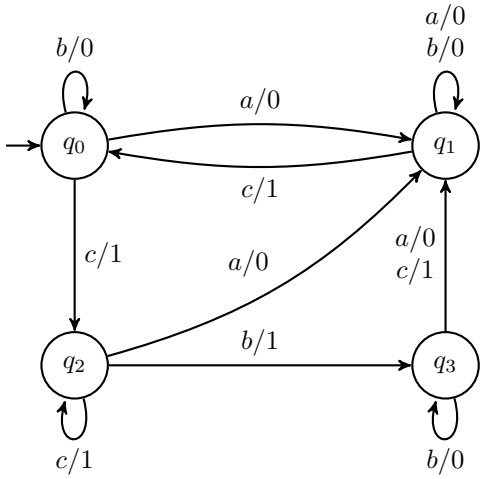


Figure 5.1: FSM M_5.

A possible minimal state cover of M_5 is given by $V_5 = \{\epsilon, a/0, c/1, cb/11\}$, which employs the following traces to reach all states of M_5:

$$\upsilon_{q_0} := \epsilon \qquad \upsilon_{q_1} := a/0 \qquad \upsilon_{q_2} := c/1 \qquad \upsilon_{q_3} := cb/11$$

Furthermore, all pairs of distinct states of M_5 can be distinguished from each other. For example, q_0 can be distinguished from q_1 and q_3 by $cb/10$ and from q_2 by $b/0$. State q_1 can be distinguished from q_2 via $b/0$ and from q_3 via $ccb/110$. Finally, q_2 and q_3 can be distinguished by $b/0$. Thus, M_5 is minimal.

5.1.2 W-Method

The W-Method [23, 115] is among oldest m-complete strategies proposed for equivalence testing. Originally defined only on completely specified DFSMs, it constructs a test suite by extending each sequence $\bar{x} \in V|_{\Sigma_I}$ for some state cover V of M with all sequences up to length $m - n + 1$ and then appends a

characterisation set W after the resulting sequence. That is, a test suite TS is constructed as

$$TS = V|_{\Sigma_I} . \left(\bigcup_{i=0}^{m-n+1} \Sigma_I^i . W|_{\Sigma_I} \right)$$

This can easily be extended to IO-based test cases, as implemented in Algorithm 2. In this algorithm, I apply a slight optimisation for partial nondeterministic FSMs by considering only extensions of V that result in combined traces whose proper prefixes are in $\mathcal{L}(M)$. Furthermore, the characterisation set is appended only after traces in $\mathcal{L}(M)$, that is, only after traces that actually reach a state in M. These optimisations are justified by Lemma 3.2.3.

Algorithm 2: W-Method

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
choose a state cover V of M
choose a characterisation set W of M
TS ← short_M(V.(⋃_{i=0}^{m−n+1} (Σ_I × Σ_O)^i))
foreach α ∈ short_M(V.(⋃_{i=0}^{m−n+1} (Σ_I × Σ_O)^i)) do
    if α ∈ L(M) then
        TS ← TS ∪ {α}.W
return TS
```

Test suites computed using the W-Method satisfy the H-Condition described in Lemma 4.1.3, as the W-Method appends the same characterisation set W after all pairs of traces in $V.(\bigcup_{i=0}^{m-n+1}(\Sigma_I \times \Sigma_O)^i) \cap \mathcal{L}(M)$ and hence ensures that all pairs required by the H-Condition are extended by a shared separating trace. Note, however, that a characterisation set usually contains many more traces than required to distinguish a single pair of states, and thus that the W-Method generates test suites containing many more test cases than required by the H-Condition. This is visualised in Fig. 5.2, which depicts a subset of a test suite generated by the W-Method, where $\upsilon_q, \upsilon_{q'}$ are the state cover traces reaching q and q' in the reference model M, respectively, and $\alpha.\alpha'$ is an extension of length at most $m-n+1$ applied after υ_q. In particular, Fig. 5.2 visualises how $\upsilon_q.\alpha.\alpha'$ is separated from other traces if pairs $(\upsilon_q.\alpha', \upsilon_q.\alpha.\alpha')$ and $(\upsilon_{q'}, \upsilon_q.\alpha.\alpha')$ are required to be separated by the H-Condition. This is performed in the W-Method by applying characterisation set W after $\upsilon_q.\alpha.\alpha'$, $\upsilon_q.\alpha$, and $\upsilon_{q'}$. By definition, W must contain some trace that distinguishes q-after-$\alpha.\alpha'$ from q-after-α, as well as some trace distinguishing q-after-$\alpha.\alpha'$ from q'. In Fig. 5.2, the first of these is visualised by a green line, while the second trace is visualised as an orange line[1]. As W distinguishes all pairs of states in M, it may also include

[1]Distinguishing traces in Fig. 5.2 and subsequent visualisations of test strategies are distinguishable not only by colour but also by line shape as described in the legend.

traces not necessary to apply after $\upsilon_q.\alpha.\alpha'$ as they serve only to distinguish other states than q-after-$\alpha.\alpha'$. Fig. 5.2 visualises this by highlighting that set W applied after $\upsilon_q.\alpha.\alpha'$ may include a blue line not employed in distinguishing q-after-$\alpha.\alpha'$.

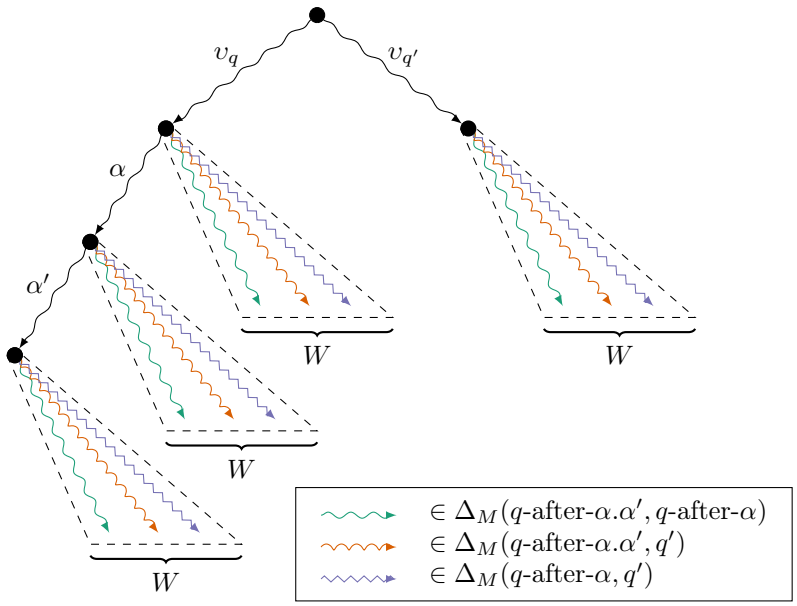


Figure 5.2: Visualisation of the W-Method separating some $\upsilon_q.\alpha.\alpha'$ as required by the H-Condition.

Consider, the running example M_5 (Fig. 5.1). A possible characterisation set of M_5 consisting only of minimum length distinguishing traces is $W = \{b/0, cb/10, ccb/110\}$. Applying the W-Method for $m = 4$ thus appends W after every trace $\upsilon_q.\alpha \in \mathcal{L}(M)$ where $|\alpha| \leq m-n+1 = 1$. That is, the resulting test suite extends each trace in state cover $V_5 = \{\epsilon, a/0, c/1, cb/11\}$ of M_5 with

$$\bigcup_{i=0}^{m-n+1} (\Sigma_I \times \Sigma_O)^i = \{\epsilon\} \cup (\Sigma_I \times \Sigma_O)^1 = \{\epsilon, a/0, a/1, b/0, b/1, c/0, c/1\}$$

and appends W after each resulting trace that is in $\mathcal{L}(M_5)$. The obtained test suite contains 39 maximal IO-traces, which constitute 27 distinct input sequences after removal of proper prefixes. An example of a superfluous trace occurs for $\upsilon_{q_2}.(c/1) = cc/11$, which reaches q_2 in M_5 and is extended with the entire characterisation set W, while it would suffice to append only $b/0$ or $b/1$, which each distinguish q_2 from all other states of M_5.

5.1.3 Wp-Method

The Wp-Method introduced in [35] and generalised to nondeterministic FSMs in [71] improves upon the W-Method by reducing the number of traces applied to extensions of $V|_{\Sigma_I}$ with sequences of length exactly $m-n+1$. Instead of the full characterisation set, these sequences $\bar{x}$ are extended only with a subset of the characterisation set that is sufficient to distinguish the reached state from all other states of M. Thus, a test suite is constructed as a union of two parts WP_1 and WP_2 defined as follows:

$$WP_1 = V|_{\Sigma_I}.(\bigcup_{i=0}^{m-n} \Sigma_I^i).W$$

$$WP_2 = \bigcup\{\{\bar{x}\}.W_q | q \in Q \wedge \bar{x} \in V|_{\Sigma_I}.\Sigma_I^{m-n+1} \wedge q \in M\text{-after-}\bar{x}\}$$

where for each state q of M the state identifier W_q is a subset of W such that for all other states q' of M this set contains a sequence distinguishing q and q'.

Algorithm 3 describes how this approach can be lifted to possibly partial and nondeterministic OFSMs, again applying the optimisation of shortening the initial test suite for M and appending the characterisation set or state identifiers only after traces in $\mathcal{L}(M)$.

Note here that an application of the Wp-Method to nondeterministic FSMs using input sequences instead of IO-traces as test cases can result in appending superfluous state identifiers. This is a consequence of the fact that sequences in $V|_{\Sigma_I}$ may reach more than one state and hence M-after-$\bar{x}$ in WP_2 may reach states not reached by any α considered in line 10 of Algorithm 3.

Algorithm 3: Wp-Method

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

1 choose a state cover V of M
2 choose a characterisation set W of M
3 for each $q \in Q$ choose some $W_q \subseteq W$ such that for each $q' \in Q$ with $q' \neq q$ it holds that $W_q \cap \Delta_M(q, q') \neq \emptyset$
4 $TS \leftarrow short_M(V.(\bigcup_{i=0}^{m-n+1}(\Sigma_I \times \Sigma_O)^i))$
5 **foreach** $\alpha \in short_M(V.(\bigcup_{i=0}^{m-n}(\Sigma_I \times \Sigma_O)^i))$ **do**
6 **if** $\alpha \in \mathcal{L}(M)$ **then**
7 $TS \leftarrow TS \cup \{\alpha\}.W$
8 **foreach** $\alpha \in short_M(V.(\Sigma_I \times \Sigma_I)^{m-n+1})$ **do**
9 **if** $\alpha \in \mathcal{L}(M)$ **then**
10 $TS \leftarrow TS \cup \{\alpha\}.W_{M\text{-after-}\alpha}$
11 **return** TS

Completeness of test suites generated by the Wp-Method follows from the H-Conditions similarly to the W-Method, as the state identifiers added in WP_2 or line 10 of Algorithm 3 are subsets of the characterisations set W, ensuring that the pairs of traces to be considered for the H-Condition are extended with shared distinguishing traces. Fig. 5.3 visualises this analogously to Fig. 5.2, as identical constructions are used except in the case that $\upsilon_q.\alpha.\alpha'$ is of length $m-n+1$, in which case it would be extended only with $W_{q\text{-after-}\alpha.\alpha'}$, which may omit traces in characterisation set W that are not employed in distinguishing the state reached by $\upsilon_q.\alpha.\alpha'$ from other states of M_5.

This reduction can be observed when applying the Wp-Method to running example M_5 (Fig. 5.1) for $m = 4$. Consider again trace $\upsilon_{q_2}.(c/1) = cc/11$, which in the W-Method was extended by the whole characterisation set W. In the Wp-Method, only a subset $W_{q_2} \subset W$ sufficient to distinguish q_2 from all other states of M_5 is applied. A possible choice for W_{q_2} is $\{b/0\}$. Thus, the test suite computed by the Wp-Method does not contain an extension of $cc/11$ with $ccb/110 \in W$. Note, however, that $\upsilon_{q_2} = c/1$ is still extended with the entire set W, despite only $b/0$ or $b/1$ being required to distinguish q_2 from other states. Therefore, the resulting test suite still contains superfluous test cases such as $cccb/1110$. Overall, it contains 36 IO-traces which can be reduced to 24 maximal input sequences.

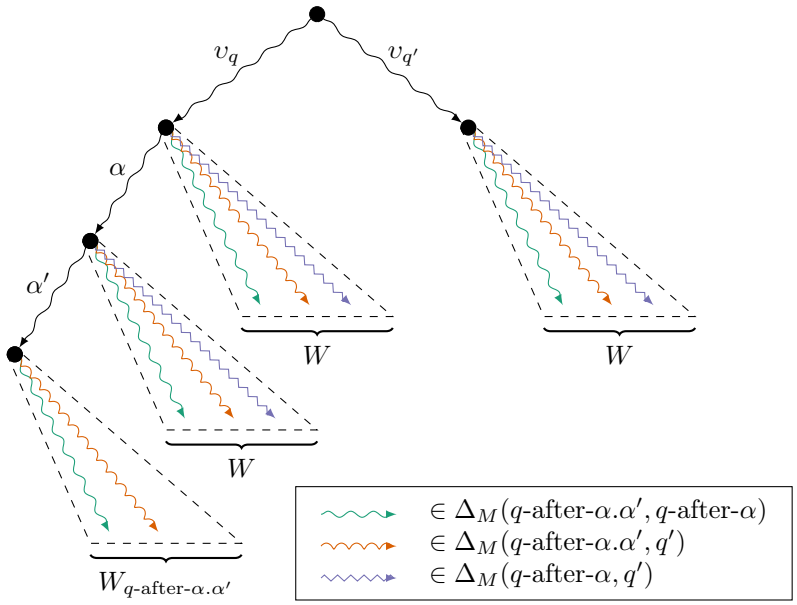


Figure 5.3: Visualisation of the Wp-Method separating some $\upsilon_q.\alpha.\alpha'$ as required by the H-Condition. Assumes that $|\alpha.\alpha'| = m-n+1$ and hence $\upsilon_q.\alpha.\alpha'$ is only extended with a state identifier instead of the entire characterisation set.

5.1.4 HSI-Method

Introduced in [72], the HSI-Method further improves upon the Wp-Method by not appending full characterisation sets and instead appending harmonised state identifiers after all $\bar{x}$ that consist of a sequence in $V|_{\Sigma_I}$ followed by an input sequence of length up to $m-n+1$. The harmonised state identifiers to append after such $\bar{x}$ again correspond to the states reached by $\bar{x}$ in M. Here, state identifiers $H_q, H_{q'}$ for distinct states q, q' are called *harmonised* if there exists a distinguishing sequence for q and q' contained (possibly as a prefix) in both H_q and $H_{q'}$. In contrast to the Wp-Method, no distinction is made between extensions of length $m-n+1$ and shorter extensions. Using the HSI-Method, a test suite TS based on input sequences can be generated for completely specified DFSM M as

$$TS := \bigcup \left\{ \{\bar{x}\}.H_q \;\middle|\; q \in Q \wedge \bar{x} \in V|_{\Sigma_I}.(\bigcup_{i=0}^{m-n+1} \Sigma_I^i) \wedge q \in M\text{-after-}\bar{x} \right\}$$

This construction can again be lifted to work with IO-traces as test cases as depicted in Algorithm 4, which employs the usual optimisation of shortening traces for M and appending harmonised state identifiers only after traces in $\mathcal{L}(M)$.

Algorithm 4: HSI-Method

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
choose a state cover V of M
for each q ∈ Q choose some H_q such that for each q' ∈ Q with q' ≠ q it
    holds that pref(H_q ∩ H_q') contains some γ ∈ Δ_M(q, q')
TS ← short_M(V.(⋃_{i=0}^{m-n+1} (Σ_I × Σ_O)^i))
foreach α ∈ short_M(V.(⋃_{i=0}^{m-n+1} (Σ_I × Σ_O)^i)) do
    if α ∈ L(M) then
        TS ← TS ∪ {α}.H_{M-after-α}
return TS
```

The m-completeness of test suites computed via the HSI-Method once more follows from the H-Conditions by similar reasoning to the W and Wp-Methods, as the state identifiers are required to be harmonised. Fig. 5.4 visualises how the separation of pairs required by the H-Condition is realised by the harmonisation of state identifiers, which ensures that for some pair $(\upsilon_q.\alpha, \upsilon_q.\alpha.\alpha')$ to be separated according to the H-Condition, state identifier $H_{q\text{-after-}\alpha.\alpha'}$ applied after $\upsilon_q.\alpha.\alpha'$ shares a trace with $H_{q\text{-after-}\alpha}$ applied after $\upsilon_q.\alpha$ that distinguishes the states reached from q with $\alpha.\alpha'$ and α. Similarly, for another pair $(\upsilon_{q'}, \upsilon_q.\alpha.\alpha')$, state identifier $H_{q\text{-after-}\alpha.\alpha'}$ applied after $\upsilon_q.\alpha.\alpha'$ shares a trace with $H_{q'}$ applied after $\upsilon_{q'}$ that distinguishes q' and the state reached from q with $\alpha.\alpha'$.

$H_{q\text{-after-}\alpha.\alpha'}$ does not, however, need to contain any traces only useful in distinguishing q' and the state reached from q with α, as was the case in the characterisation sets used in the W and Wp-Methods (see Fig. 5.2 and Fig. 5.3).

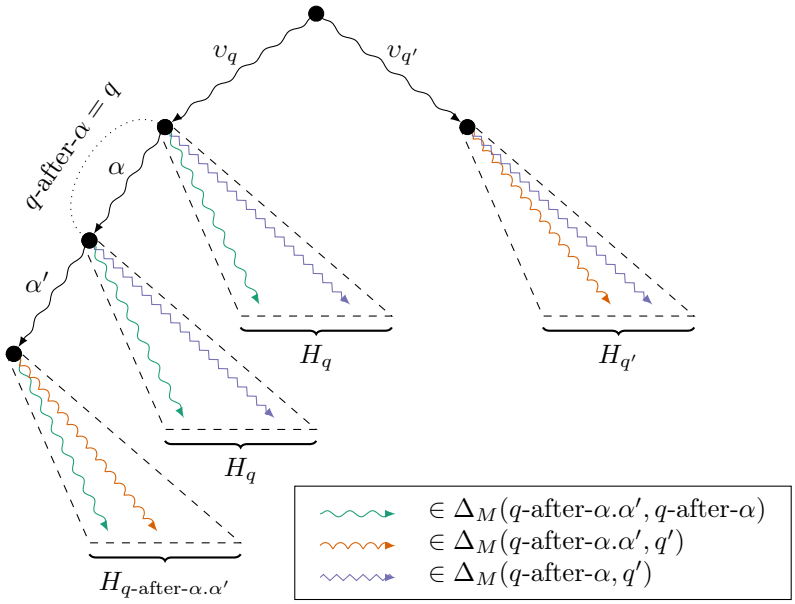


Figure 5.4: Visualisation of the HSI-Method separating some $v_q.\alpha.\alpha'$ as required by the H-Condition. Traces v_q and $v_q.\alpha$ exemplify how the exact same state identifiers are appended after convergent traces.

In applying the HSI-Method to running example M_5 (Fig. 5.1) for $m = 4$, the effect of only applying harmonised state identifiers instead of sometimes applying a full characterisation set as in the W or Wp-Methods can once more be observed for $v_{q_2} = c/1$, which in both previous test strategies is extended with characterisation set $W = \{b/0, cb/10, ccb/110\}$. Recall that $b/0$ or $b/1$ suffices to distinguish q_2 from all other states. That is, $H_{q_2} = \{b/0\}$ constitutes a state identifier of q_2. Instead of W, the HSI-Method thus applies only H_{q_2} after $v_{q_2} = c/1$ and $v_{q_2}.(c/1)$, thus never appending $cb/10$ or $ccb/110$ after $v_{q_2}.(c/1)$. Note here that the size of test suites generated using the W, Wp or HSI-Methods is strongly affected by the choice of characterisation sets and state identifiers. For M_5, a state identifier H_{q_1} could be obtained by computing for each other state q' the shortest trace distinguishing q_1 from q'. This results, for example, in $\{b/0, cb/10, ccb/110\}$, distinguishing q_1 from q_0, q_2, q_3 via $cb/10$, $b/0$, $ccb/110$, respectively. However, q_1 and q_0 can also be distinguished via $ccb/110$, which reduces the state identifier to $\{b/0, ccb/110\}$. If harmonised state identifiers are created by considering each state pair in isolation, the test suite obtained using the HSI-Method contains 35 IO-traces, which may be reduced to 23 maximal input sequences. The H-Method described in the next section

provides an alternative strategy to trying to pre-select optimal state identifiers in that it selects distinguishing traces during computation based on the current state of the test suite.

5.1.5 H-Method

All three previously presented strategies are static in that the order in which elements are added to the test suite is irrelevant as no intermediate state of the test suite during construction is queried. For example, in Algorithm 2 the same test suite is generated for each possible order of considering traces α in line 4. Thus, in appending distinguishing sequences (in the form of fixed characterisation sets or fixed state identifiers), the previously discussed strategies do not check whether the test suite computed up to that step may already extend the currently considered trace with some sequences that may be re-used to distinguish the reached state from other states.

The H-Method presented in [27] improves upon the previously discussed strategies by adding distinguishing sequences only if necessary, trying to minimise the size of the resulting test suite. Furthermore, this strategy makes more explicit the pairs of sequences to be separated. Where the previously discussed strategies effectively append distinguishing sequences after almost all pairs of sequences $\alpha, \beta \in (V|_{\Sigma_I} . \bigcup_{i=0}^{m-n+1} \Sigma_I^i)$ that reach distinct states[2], the H-Method only considers the following three sets of pairs of sequences[3]:

$$A := V|_{\Sigma_I} \times V|_{\Sigma_I}$$

$$B := V|_{\Sigma_I} \times (V|_{\Sigma_I} . \bigcup_{i=1}^{m-n+1} \Sigma_I^i)$$

$$C := \left\{ (\bar{v}\bar{x}', \bar{v}\bar{x}) \;\middle|\; \bar{v} \in V|_{\Sigma_I} \wedge \bar{x} \in \bigcup_{i=1}^{m-n+1} \Sigma_I^i \wedge \bar{x}' \in pref(\bar{x}) \setminus \{\bar{x}\} \right\}$$

From these sets, the H-Method constructs a test suite TS by initialising TS as $V|_{\Sigma_I} . \bigcup_{i=0}^{m-n+1} \Sigma_I^i$ and then iterating through all pairs $(\alpha, \beta) \in A \cup B \cup C$ whose elements reach distinct states in M. For each such (α, β), it is then checked whether the current test suite already contains some $\alpha.\gamma, \beta.\gamma'$ such that γ and γ' share a prefix that distinguishes the states reached by α and β in M. Only if this is not the case, a new sequence that distinguishes the reached states is appended to both α and β in the test suite. The choice of the above A, B, and C, directly follows from the sufficiency condition the H-Method is designed to satisfy, namely the H-Condition, whose input-based formulation is given in Lemma 4.1.1.

[2] For the W and HSI-Method, all such pairs are covered. For the Wp-Method, extensions of length $m - n + 1$ may not be separated, as the state identifiers appended after them are not required to be harmonised.

[3] The definition given in [27] differs slightly from the presentation given here, as in the presence of partial reference models the strategy presented in [27] tests for quasi-equivalence instead of language-equivalence. Both definitions coincide for completely specified DFSMs.

The order in which to consider the pairs in $A \cup B \cup C$ as well as the choice of distinguishing sequences to add allow for various applications of heuristics with the goal of generating fewer and shorter test cases. A detailed exemplary implementation is described in [106][4].

A simple lifting of the H-Method to IO-traces as test cases is presented in Algorithm 5, which collects all pairs of sequences to append with distinguishing traces in a set D filtered from $A \cup B \cup C$. The H-Method is a direct implementation of the H-Condition and thus the m-completeness of test suites computed using this method follows from the H-Condition. To simplify the presentation at this point, however, Algorithm 5 does not incorporate all refinements of Lemma 4.1.3. Improved implementations are discussed during the development of frameworks in Chapter 6.

Algorithm 5: H-Method

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

choose a state cover V of M
$A \leftarrow V \times V$
$B \leftarrow V \times (V.\bigcup_{i=1}^{m-n+1}(\Sigma_I \times \Sigma_O)^i)$
$C \leftarrow \{(v.\omega', v.\omega) \mid v \in V \wedge \omega \in \bigcup_{i=1}^{m-n+1}(\Sigma_I \times \Sigma_O)^i \wedge \omega' \in pref(\omega) \setminus \{\omega\}\}$
$D \leftarrow \{(\alpha, \beta) \in A \cup B \cup C \mid \alpha, \beta \in \mathcal{L}(M) \wedge M\text{-after-}\alpha \neq M\text{-after-}\beta\}$
$TS \leftarrow short_M(V.(\bigcup_{i=0}^{m-n+1}(\Sigma_I \times \Sigma_O)^i))$
foreach $(\alpha, \beta) \in D$ **do**
    if *there does not exist any* γ *distinguishing* M*-after-*α *and* M*-after-*β *such that* $\alpha.\gamma, \beta.\gamma \in TS$ **then**
        choose some γ that distinguishes M-after-α and M-after-β
        $TS \leftarrow TS \cup \{\alpha.\gamma, \beta.\gamma\}$
return TS

Fig. 5.5 visualises how the H-Method does not require appending the exact same distinguishing trace whenever the same pair of states needs to be distinguished according to the H-Condition. More precisely, Fig. 5.5 considers pairs of traces $(v_q.\alpha, v_q.\alpha.\alpha')$, $(v_{q'}, v_q.\alpha)$, and $(v_{q'}, v_q.\alpha.\alpha')$ to be separated according to the H-Condition, and assumes that both v_q and $v_q.\alpha$ reach q. Then $v_q.\alpha.\alpha'$ needs to be extended with some trace to separate it from $v_q.\alpha$, such as γ_1. Furthermore, $v_q.\alpha.\alpha'$ needs to be separated from v_q, for which purpose the same γ_1 could be used. However, if in the test suite computed so far v_q has already been extended with some γ_2 that also distinguishes q from q-after-$\alpha.\alpha'$, then it may result in a smaller test suite to instead append γ_2 after $v_q.\alpha.\alpha'$. Similarly, different traces γ_4 and γ_5 may be appended to $v_q.\alpha$ and v_q in order

[4] For practical implementations see, for example, [105] and the `libfsmtest` open source library described in [4].

to distinguish q from q' reached by $\upsilon_{q'}$. This highlights the greater flexibility of the H-Method compared to previously discussed strategies with respect to choosing distinguishing traces (see Figures 5.2, 5.3, and 5.4).

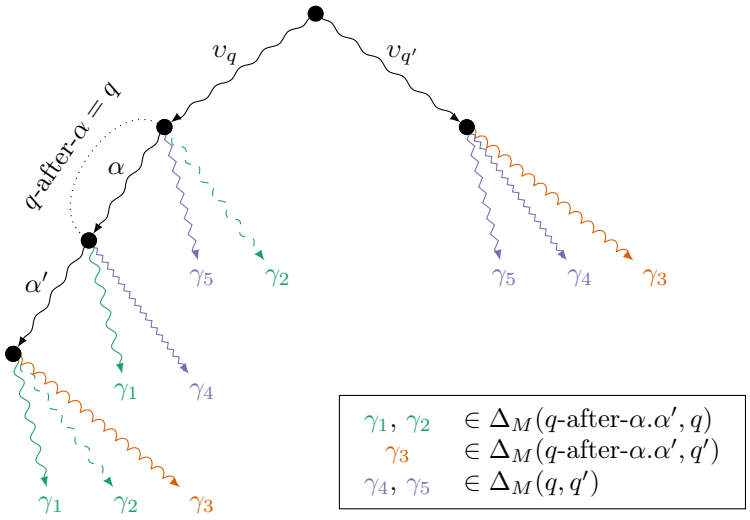


Figure 5.5: Visualisation of the H-Method separating some $\upsilon_q.\alpha.\alpha'$ as required by the H-Condition. Note how different traces may be used to distinguish the same pairs of states, such as γ_1 and γ_2 which distinguish the state reached by $\upsilon_q.\alpha.\alpha'$ from q (reached by both υ_q and $\upsilon_q.\alpha$).

Applying the H-Method to running example M_5 (Fig. 5.1) for $m = 4$, the H-Method may, for example, arrive at the situation where the trace pair $(\upsilon_{q_1}, \upsilon_{q_3}.(b/0)) = (a/0, cbb/110)$ needs to be separated in line 7 of Algorithm 5 and the test suite already contains $ab/00$ and $cbbb/1100$. The shortest distinguishing trace for the pair is $ccb/110$ or $ccb/111$, so it would be possible to simply append either of them after the pair. However, it would also be possible to extend $ab/00$ and $cbbb/1100$ that are already in the test suite with $ccb/110$, as $bccb/0110$ also is a distinguishing trace for the pair. If $cbbb/1000$ currently is a maximal trace in the test suite, this latter choice might be preferable, as it avoids branching in the test suite, that is, it avoids having two distinct maximal test cases $cbbb/1100, cbbccb/110110$, instead resulting in a single trace $cbbbccb/1100110$. The same considerations apply to $ab/00$. Section 6.3 shows how Algorithm 13 introduced below for the SPYH-Method (see Subsection 5.1.7) may be employed to implement this heuristic to search for suitable distinguishing traces. Applying this implementation of the H-Method to M_5 for $m = 4$ generates a test suite containing 33 IO-traces which can be reduced to 19 maximal input sequences. A further example of applying the H-Method is given in Appendix B, which compares a naive application of the H-Method using input sequences with that of using IO-traces.

5.1.6 SPY-Method

The SPY-Method developed in [102, 103] is the first strategy to make explicit use of convergence of sequences in order to distribute distinguishing sequences to obtain smaller test suites. That is, during computation of a test suite it establishes that if a system under test passes the current test suite, then certain sequences converge in both the reference model and the SUT. Thereafter, if distinguishing sequence γ is to be appended after some α that is known to converge with β, the method may instead append γ after β, if this results in fewer or shorter test cases. In the selection of distinguishing traces to append, the SPY-Method uses harmonised state identifiers and hence can be seen as an improvement upon the HSI-Method rather than the H-Method.

Algorithm 6 describes how the SPY-Method computes a test suite TS for minimal complete DFSMs[5]. Similar to the HSI-Method, it first computes a state cover V (line 1) and a family of harmonised state identifiers (line 2), which are then appended after the sequences in the state cover, corresponding to the reached state in M (line 3). Next, a convergence graph is initialised with this intermediate test suite (line 4), which is to be used in later steps to store information about sequences that converge in all complete DFSMs in $\mathcal{F}(M, m)$ passing the resulting test suite. Thereafter, all *unverified* transitions are considered in the loop beginning in line 5, where a transition (q, x, y, q') of M is *verified* if the sequence reaching q' in V is equal to the sequence reaching q in V followed by input x, that is, if $\bar{v}_{q'} = \bar{v}_q.x$. For each unverified transition (q, x, y, q'), the SPY-Method then considers all sequences $\bar{u} \in \Sigma_I^*$ of length up to $m - n$ (line 6) and distributes the harmonised state identifier $H_{q'\text{-after-}\bar{u}}$ of the state reached in M from q' by $\bar{u}$ both over sequences convergent with $\bar{v}_q$ followed by x, and sequences convergent with $\bar{v}_{q'}$ (lines 7 to 9). The handling of (q, x, y, q') concludes by marking $\bar{v}_q.x$ and $\bar{v}_{q'}$ as convergent in the convergence graph G (line 10), which also merges all $\bar{v}_q.x.\bar{x}$ and $\bar{v}'_q.\bar{x}$ contained in the test suite (see property (1.) of Lemma 2.5.2).

Thus, instead of iterating through a set of pairs of sequences to append with distinguishing sequences as in the H-Method, the SPY-Method seeks to satisfy the SPY-Condition for completely specified DFSMs (analogous to Corollary 4.2.2) by establishing that the resulting test suite TS contains a subset A with $\epsilon \in A$ that is a convergence-preserving state cover of M for $\mathcal{F}(M, m)$. That is, for each transition (q, x, y, q') of M it must be ensured that $\bar{v}_q.x$ and $\bar{v}_{q'}$ are convergent in all $I \in \mathcal{F}(M, m)$ that pass TS. This is realised in two steps. First, for transitions where $\bar{v}_q.x$ and $\bar{v}_{q'}$ coincide (that is, those transitions not considered in line 5), no further action is necessary, as the corresponding sequences are identical. Second, each remaining transition is verified in the loop beginning in line 5 by satisfying the sufficient condition for convergence on completely

[5] Here and in the subsequent subsection I initially apply functions on convergence graphs (see Section 3.3) to input sequences instead of IO-traces in a slight abuse of notation. As described in [103], input sequences converge in a complete DFSM if their unique corresponding IO-traces converge. As only input sequences over the input alphabet of the reference model are considered, it is also possible to omit checks for language containment.

specified DFSMs, which is analogous to Lemma 4.2.5 (see Lemma 3 of [103] for a version restricted to completely specified DFSMs), by extending convergent sequences of both $\bar{v}_q.x$ and $\bar{v}_{q'}$ with sequences up to length $m-n$ and applying harmonised state identifiers to establish preservation of divergence. After handling each pair of $\bar{v}_q.x$ and $\bar{v}_{q'}$, it is thus valid to merge their convergent classes in line 10, allowing later iterations more choices to distribute over convergent sequences.

Note here that the SPY-Method still effectively considers extensions after the state cover of length up to $m-n+1$, as the x in $\bar{v}_q.x$ already extends $\bar{v}_q$ by a single input.

Algorithm 6: SPY-Method using input sequences as test cases

Input : minimal complete DFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq \Sigma_I^*$

1 choose a minimal state cover V of M
2 for each $q \in Q$ choose some H_q such that for each $q' \in Q$ with $q' \neq q$ it holds that $pref(H_q \cap H_{q'})$ contains some γ that distinguishes q and q'
3 $TS \leftarrow \bigcup_{\bar{x} \in V|_{\Sigma_I}} \{\bar{x}\}.H_{M\text{-after-}\bar{x}}$
4 CG-INITIALISE(TS)
5 **foreach** $(q, x, y, q') \in h$ *not already verified by* TS **do**
6 **foreach** $\bar{u} \in \bigcup_{i=0}^{m-n} \Sigma_I^i$ **do**
7 **foreach** $w \in H_{q'\text{-after-}\bar{u}}$ **do**
8 DISTRIBUTEEXTENSION(M, $\bar{v}_q$, $x.\bar{u}\bar{w}$, TS, G)
9 DISTRIBUTEEXTENSION(M, $\bar{v}_{q'}$, $\bar{u}\bar{w}$, TS, G)
10 CG-MERGE($G, \bar{v}_q.x, \bar{v}_{q'}$)
11 **return** TS

Similar to the H-Method, the above implementation of the SPY-Method contains several steps where heuristics may be applied to try to obtain small test suites, including the order in which transitions are considered and the criteria by which DISTRIBUTEEXTENSION chooses which convergent sequence to extend.

To my best knowledge, the SPY-Method has not previously been generalised to possibly partial and nondeterministic OFSMs. Algorithm 7 introduces an exemplary implementation of this generalisation by modifying Algorithm 6 in two aspects. First, similarly to the algorithms for previously discussed strategies, shortened IO-traces are used instead of input sequences and only traces in $\mathcal{L}(M)$ are extended with harmonised state identifiers (lines 6 to 13). Second, the transition cover is completed by appending for each state q of M every IO-pair x/y not exhibited by $\mathcal{L}_M(q)$ to some trace converging with the trace reaching q in the chosen state cover (lines 15 and 16). The need for this latter step in lifting the SPY-Method to IO-traces follows from the SPY-Condition (Corollary 4.2.2), where a transition cover requires covering every combination (q, x, y) of states q,

inputs x, and outputs y of M. When considering only complete DFSMs as reference models and input sequences as test cases, this is not necessary, as in this case only state-input pairs are to be considered and in a complete DFSM there exists one transition from each state for each input, thus covering all required pairs during the verification of all transitions.

Algorithm 7: SPY-Method lifted to IO-traces as test cases

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

choose a state cover V of M
for each $q \in Q$ choose some H_q such that for each $q' \in Q$ with $q' \neq q$ it holds that $pref(H_q \cap H_{q'})$ contains some γ that distinguishes q and q'
$TS \leftarrow \bigcup_{\alpha \in V} \{\alpha\}.H_{M\text{-after-}\alpha}$
CG-INITIALISE(TS)
foreach $(q, x, y, q') \in h$ *not already verified by* TS **do**
    foreach $\omega \in \bigcup_{i=0}^{m-n} (\Sigma_I \times \Sigma_O)^i$ *such that* $pref(\omega) \setminus \{\omega\} \subseteq \mathcal{L}_M(q')$ **do**
        if $\omega \in \mathcal{L}_M(q')$ **then**
            foreach $\gamma \in H_{q'\text{-after-}\omega}$ **do**
                DISTRIBUTEEXTENSION(M, υ_q, $(x/y).\omega.\gamma$, TS, G)
                DISTRIBUTEEXTENSION(M, $\upsilon_{q'}$, $\omega.\gamma$, TS, G)
        else
            DISTRIBUTEEXTENSION(M, υ_q, $(x/y).\omega$, TS, G)
            DISTRIBUTEEXTENSION(M, $\upsilon_{q'}$, ω, TS, G)
    CG-MERGE($G, \upsilon_q.(x/y), \upsilon_{q'}$)
foreach $q \in Q, x \in \Sigma_I, y \in \Sigma_O$ *such that* $x/y \notin \mathcal{L}_M(q)$ **do**
    DISTRIBUTEEXTENSION(M, υ_q, x/y, TS, G)
return TS

Completeness of test suites generated by the SPY-Method can be established by the SPY-Condition (Corollary 4.2.2) but also by directly employing the S-Condition (Lemma 4.1.5), as the SPY-Method separates pairs of traces described in the S-Condition essentially in the same way as the HSI-Method[6], augmented by distribution over convergent traces. Fig. 5.6 visualises how the S-Condition is satisfied by distributing harmonised state identifiers, where the distribution over convergent traces is the essential difference to the HSI-Method visualised in Fig. 5.4.

Application of the SPY-Method to running example M_5 (Fig. 5.1) exemplifies how distribution over convergent traces can be employed to reduce test suite size. This application first appends each $\upsilon_q \in V_5$ with harmonised state identifier H_q of the reached state. Thereafter, it is checked which transitions

[6] As discussed in Section 4.1.2, the S-Condition coincides with the H-Condition if distribution over convergent traces is not used in test suite generation.

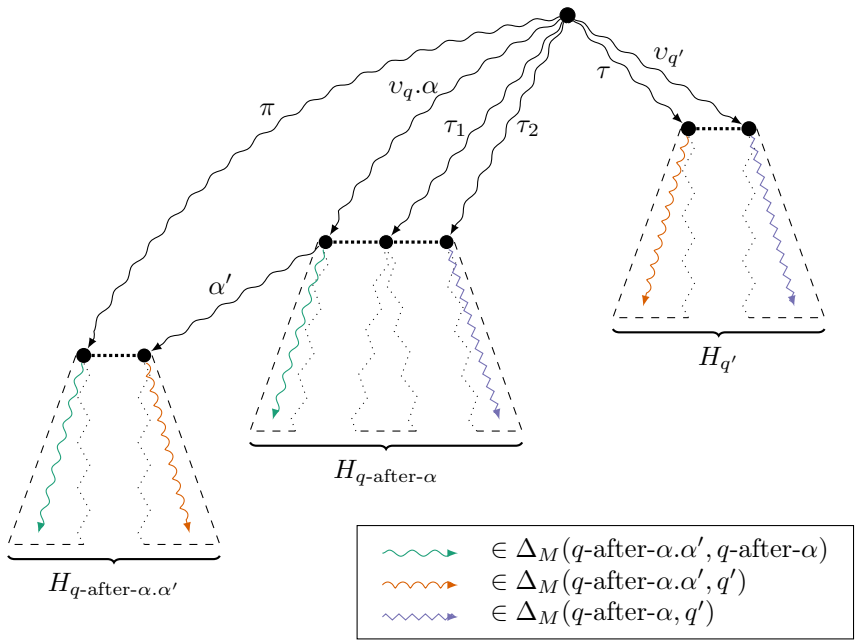


Figure 5.6: Visualisation of the SPY-Method separating some $\upsilon_q.\alpha.\alpha'$ as required by the S-Condition. Dotted lines between nodes indicate that the traces reaching these nodes are $\mathcal{F}(M,m)_{TS}$-convergent. Note how harmonised state identifiers are distributed over $\mathcal{F}(M,m)_{TS}$-convergent traces.

of M_5 are already verified by the choice of transition cover. For V_5, these are $(q_0, a, 0, q_1)$, $(q_0, c, 1, q_2)$ and $(q_2, b, 1, q_3)$, since $\upsilon_{q_0}.(a/0) = \upsilon_{q_1}$, $\upsilon_{q_0}.(c/1) = \upsilon_{q_2}$ and $\upsilon_{q_2}.(b/1) = \upsilon_{q_3}$, respectively. Then the remaining transitions are handled. Up to this point, no convergences have been established. When handling, for example, transition $(q_1, c, 1, q_0)$ first, then $\{c/1\}.H_{q_0}$ and H_{q_0} are applied after $\upsilon_{q_1} = a/0$ and $\upsilon_{q_0} = \epsilon$, respectively, as for $m = 4$ the only choice for ω in line 6 of Algorithm 7 is ϵ and hence $H_{q_0\text{-after-}\omega} = H_{q_0}$. This establishes the convergence of $\upsilon_{q_1}.(c/1) = ac/01$ and $\upsilon_{q_0} = \epsilon$, which may be exploited in the handling of later transitions. Let $H_{q_0} = \{b/0, cb/10\}$. Then the test suite in particular contains $acb/010$. Suppose transition $(q_0, b, 0, q_0)$ is to be handled next. Then $b/0.H_{q_0}$ and H_{q_0} are both to be distributed over $[\upsilon_{q_0}]_{TS}$. Consider $bb/00 \in \{b/0\}.H_{q_0}$ first. As the test suite already contains $b/0$, it is suitable to append $b/0$ after $\epsilon \in [\upsilon_{q_0}]_{TS}$, effectively replacing test case $b/0$ with $bb/00$. Consider next $bcb/010 \in H_{q_0}$. Appending this trace after ϵ would cause branching in the test suite due to containment of $bb/00$. Appending $bcb/010$ after $ac/01 \in [\upsilon_{q_0}]_{TS}$, however, might not induce branching, as $acbcb/01010$ is an extension of $acb/010$. Thus, instead of adding a new branch in the test suite, this latter option may merely extend a test case already in the test suite, thus avoiding a potentially costly reset operation when applying the test suite in practice. Following this concept, the SPY-Method creates a test suite consisting of 25 IO-traces that may be reduced to 17 maximal input sequences.

5.1.7 SPYH-Method

Detailed in the previous two sections, the H and SPY-Methods improve upon the HSI-Method in two different aspects, as the former allows for dynamically choosing distinguishing traces instead of fixed harmonised state identifiers, whereas the latter allows for dynamically distributing fixed harmonised state identifiers over convergent traces. The SPYH-Method introduced in [108] combines these approaches by performing both the selection and the distribution of distinguishing traces in a dynamic way.

Similar to the SPY-Method, the SPYH-Method has to my best knowledge only been proposed for complete DFSMs and using input sequences as test cases. Algorithm 8 provides an implementation of this original description[7], following the implementation given in [106, 108]. The SPYH-Method begins similar to the SPY-Method by computing a state cover V, but does not require computation of harmonised state identifiers. Thus, the test suite and the convergence graph are initialised just by the state cover and then method SPYH-DISTINGUISH is employed on each sequence $\bar{v}$ in the state cover to make it TS-separable from other sequences in V (lines 4 and 5). Instead of applying harmonised state identifiers, this method, implemented in Algorithm 9, proceeds to separate $\bar{v}$ from all other sequences $\bar{v}'$ in the state cover by finding a separating sequence and distributing it over sequences converging with $\bar{v}$ and $\bar{v}'$, respectively. In doing so it

[7] Analogous to Algorithm 6 and Subsection 5.1.6, this implementation applies operations on convergence graphs to input sequences instead of IO-traces.

employs function BESTPREFIXOFSEPSEQ, which returns a pair $(e, \bar{w}) \in \mathbb{N} \times \Sigma_I^*$ such that if e is 0, then the test suite already separates $\bar{v}$ and $\bar{v}'$. If $e > 0$, then $\bar{w}$ is a prefix of a distinguishing trace of the states reached by $\bar{v}$ and $\bar{v}'$ and SPYH-DISTINGUISH chooses a distinguishing trace for $\bar{v}$ and $\bar{v}'$ by extending $\bar{w}$. For succinctness, I here omit a full description of BESTPREFIXOFSEPSEQ for testing based on input sequences – a full implementation for the IO-based approach is given in Algorithm 13 and an implementation for complete DFSMs can be found in Algorithm 24 of [106].

Next, the SPYH-Method filters unverified transitions analogous to the SPY-Method and then sorts them in ascending order by assigning to each transition (q, x, y, q') as value the sum of the length of the sequences reaching q and q' in the state cover (line 6). This ordering serves to establish convergences of states "closer" to the initial state first, as they are visited along traces reaching more "distant" states and hence such convergences are more likely to be useful in obtaining smaller test suites.

Following this sorting, the SPYH-Method iteratively handles each unverified transition (q, x, y, q') (beginning in line 7) and establishes preservation of the convergence of $\bar{v}_q.x$ and $\bar{v}_{q'}$ by satisfying the sufficient condition of Lemma 4.2.5 (again see Lemma 3 of [103] for a version restricted to DFSMs). This is performed by DISTINGUISHFROMSET$_I$, implemented in Algorithm 10. This recursive function extends $\bar{v}_q.x$ and $\bar{v}_{q'}$ with all sequences of length up to $m - n$ and separates sequences as necessary to satisfy Lemma 4.2.5. More precisely, in each recursion step of DISTINGUISHFROMSET$_I$, a current extension $\bar{x}$ of both $\bar{v}_q.x$ and $\bar{v}_{q'}$ is considered (initially ϵ) and $\bar{v}_q.x.\bar{x}, \bar{v}_{q'}.\bar{x}$ are separated from all diverging sequences in $V|_{\Sigma_I} \cup \{\bar{v}_q.x, \bar{v}_{q'}\}.pref(\bar{x})$ (lines 1 to 4 of DISTINGUISHFROMSET$_I$), where the latter set is stored in variable X. Then, if $\bar{x}$ has not reached length $m - n$ (line 5 of DISTINGUISHFROMSET$_I$, k is decreased in each recursive step), all extensions $\bar{x}.x'$ for $x' \in \Sigma_I$ are considered by first appending them to $\bar{v}_q$ and $\bar{v}_{q'}$ and then performing the next recursive step (lines 9 to 12 of DISTINGUISHFROMSET$_I$). Lines 6 to 8 and 13 to 15 of DISTINGUISHFROMSET$_I$ serve to update and keep free of superfluous convergent sequences the set $V|_{\Sigma_I} \cup \{\bar{v}_q.x, \bar{v}_{q'}\}.pref(\bar{x})$, which is passed to recursive calls. Having established preservation of convergence of $\bar{v}_q.x$ and $\bar{v}_{q'}$, the SPYH-Method then merges finally these sequences in line 12 of Algorithm 8, which enables later iterations to distribute sequences over the unified convergent classes of $\bar{v}_q.x$ and $\bar{v}_{q'}$ and their successors (see property (1.) of Lemma 2.5.2).

Generalising the SPYH-Method to partial nondeterministic FSMs and IO-traces as test cases requires modifications similar to the generalisation of the SPY-Method presented in the previous subsection. A possible implementation is given in Algorithm 11, which differs from Algorithm 8 in particular in the use of IO-traces, the corresponding use of DISTINGUISHFROMSET instead of DISTINGUISHFROMSET$_I$, and the loop in lines 13 and 14 that ensures that no 3-tuple of state, input and output is missed in creating a transition cover of M.

Function DISTINGUISHFROMSET (Algorithm 12) differs from the previously described DISTINGUISHFROMSET$_I$ (Algorithm 10) in the use of IO-traces and in the inclusion of an optimisation for partial reference models (lines 11,12) that

Algorithm 8: SPYH-Method using input sequences as test cases

Input : minimal complete DFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq \Sigma_I^*$

choose a minimal state cover V of M
$TS \leftarrow V|_{\Sigma_I}$
CG-INITIALISE(TS)
foreach $\bar{x} \in V$ **do**
    SPYH-DISTINGUISH($\bar{x}, V, TS, G$)
sort unverified transitions in ascending order by weight function $(q, x, y, q') \mapsto |\bar{v}_q| + |\bar{v}_{q'}|$
foreach $(q, x, y, q') \in h$ *not already verified by* TS **do**
    $TS \leftarrow TS \cup \{\bar{v}_q.x\}$
    insert $\bar{v}_q.x$ to G
    $S \leftarrow V|_{\Sigma_I}$
    DISTINGUISHFROMSET$_I(\bar{v}_q.x, \bar{v}_{q'}, V, S, TS, G, m - n)$
    CG-MERGE$(G, \bar{v}_q.(x/y), \bar{v}_{q'})$
return TS

Algorithm 9: SPYH-DISTINGUISH(α, V, TS, G)

foreach $\beta \in V$ *such that* M*-after-*$\alpha \neq M$*-after-*β **do**
    $(e, \gamma) \leftarrow$ BESTPREFIXOFSEPSEQ(α, β, TS, G)
    if $e > 0$ **then**
        choose some $\gamma' \in \Delta_M(M\text{-after-}\alpha.\gamma, M\text{-after-}\beta.\gamma)$
        DISTRIBUTEEXTENSION(M, α, $\gamma.\gamma'$, TS, G)
        DISTRIBUTEEXTENSION(M, β, $\gamma.\gamma'$, TS, G)

Algorithm 10: distinguishFromSet$_I(\bar{u}, \bar{v}, V, X, TS, G, k)$

```
1  SPYH-distinguish(ū, X, TS, G)
2  notReferenced ← ∄q ∈ Q. v̄_q ∈ CG-lookup(G, v̄)
3  if notReferenced then
4      SPYH-distinguish(v̄, X, TS, G)
5  if k > 0 then
6      push ū to X
7      if notReferenced then
8          push v̄ to X
9      foreach x ∈ Σ_I do
10         distributeExtension(M, ū, x, TS, G)
11         distributeExtension(M, v̄, x, TS, G)
12         distinguishFromSet_I(ū.x, v̄.x, V, X, TS, G, k − 1)
13     if notReferenced then
14         pop v̄ from X
15     pop ū from X
```

Algorithm 11: SPYH-Method lifted to IO-traces as test cases

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Output: an m-complete test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
1  choose a minimal state cover V of M
2  TS ← V
3  CG-initialise(TS)
4  foreach v ∈ V do
5      SPYH-distinguish(v, V, TS, G)
6  sort unverified transitions in ascending order by weight function
       (q, x, y, q′) ↦ |v_q| + |v_q′|
7  foreach (q, x, y, q′) ∈ h not already verified by TS do
8      TS ← TS ∪ {v_q.(x/y)}
9      insert v_q.(x/y) to G
10     S ← V
11     distinguishFromSet(v_q.(x/y), v_q′, V, S, TS, G, m − n)
12     CG-merge(G, v_q.(x/y), v_q′)
13 foreach q ∈ Q, x ∈ Σ_I, y ∈ Σ_O such that x/y ∉ L_M(q) do
14     distributeExtension(M, v_q, x/y, TS, G)
15 return TS
```

Algorithm 12: distinguishFromSet$(\alpha, \beta, V, X, TS, G, k)$

```
1  SPYH-distinguish(α, X, TS, G)
2  notReferenced ← ∄q ∈ Q. v_q ∈ CG-lookup(G, β)
3  if notReferenced then
4      SPYH-distinguish(β, X, TS, G)
5  if k > 0 then
6      push α to X
7      if notReferenced then
8          push β to X
9      foreach x ∈ Σ_I, y ∈ Σ_O do
10         distributeExtension(M, α, x/y, TS, G)
11         distributeExtension(M, β, x/y, TS, G)
12         if α.(x/y) ∈ L(M) then
13             distinguishFromSet(α.(x/y), β.(x/y), V, X, TS, G, k − 1)
14     if notReferenced then
15         pop β from X
16     pop α from X
```

performs the recursive step only for extensions in the language of the reference model M (recall that Lemma 4.2.5 only requires extensions whose proper prefixes are in $\mathcal{L}(M)$). This optimisation is analogous to the shortening performed in previous methods.

Finally, function bestPrefixOfSepSeq for partial nondeterministic OFSMs is implemented in detail in Algorithm 13. For inputs α, β, this function returns a pair (*minEstimate*, *bestPrefix*), where *bestPrefix* is the prefix to some heuristically chosen distinguishing trace of the states reached in M by α and β that should be added to the test suite in order to separate α and β. The second value in the pair, *minEstimate*, represents an estimated cost of appending this distinguishing trace to the test suite. If *minEstimate* is 0, then α and β are already separable by the current test suite and the test suite does not need to be modified. The value of *minEstimate* may exceed the length of the considered distinguishing trace if appending it adds a new branch to the test suite. Function bestPrefixOfSepSeq begins by obtaining minimal convergent traces π and τ for α and β, respectively, storing the states reached by π and τ in M as q_π and q_τ, respectively. Then, *minEstimate* is initialised with the length of the shortest sequence distinguishing q_π and q_τ and *bestPrefix* is initialised with the empty trace (lines 1 to 6). If no traces convergent with π or τ are maximal in the current test suite, then in lines 7 to 10 their respective lengths are added to *minEstimate*, as extending them would cause adding new branches in the tree representing test suite. Maximality in the test suite is tested by function hasLeaf (Algorithm 14), which checks for a given trace γ whether any trace in $[\gamma]_{TS}$ is maximal in the test suite TS. After these

initialisations, BESTPREFIXOFSEPSEQ iterates through all IO-pairs (line 11). If for some $x/y \in \Sigma_I \times \Sigma_O$ the test suite (including prefixes) contains traces $\pi'.(x/y), \tau'.(x/y)$ where π' and τ' are convergent with π and τ, respectively, then the block beginning in line 14 is entered. Here it is first checked whether x/y already distinguishes the states reached by π and τ, in which case $(0, \epsilon)$ is returned. If x/y reaches the same state after both q_π and q_τ, then this IO-pair does not need to be considered further, as it cannot be prefix to a distinguishing trace (line 15). Otherwise, BESTPREFIXOFSEPSEQ is recursively applied to $\pi.(x/y)$ and $\tau.(x/y)$ and either this results in finding that the test suite does not need to be extended (line 17), or the current values of (*minEstimate*, *bestPrefix*) are updated if a smaller estimate is observed (lines 18, 19). If x/y extends a convergent trace of only either π or τ, then the blocks starting in line 21 or line 30 are entered, respectively. Each of these estimates the length of a distinguishing trace to be distributed after both π and τ and updates *minEstimate* based on further maximality considerations, possibly also updating *bestPrefix*. These estimations employ function ESTIMATEGROWTH (Algorithm 16), which returns just 1 if x/y distinguishes q_π and q_τ, as x/y is already applied after a convergent trace of either π or τ and hence the test suite needs to be extended by only a single IO-pair. If x/y takes q_π and q_τ to either the same state or π, τ reach the same states as $\pi.(x/y), \tau.(x/y)$, then $2|Q|$ is returned, which is longer than any minimal distinguishing trace in M. Otherwise, a shortest trace γ for q-after-x/y and q'-after-x/y is obtained and ESTIMATEGROWTH returns $2|\gamma| + 1$. In the final case of considering x/y in BESTPREFIXOFSEPSEQ, if x/y extends no convergent trace of π or τ, then x/y is not considered further. All checks whether a trace convergent to a certain given trace are extended by x/y in the test suite are implemented as calls to helper function HASEXTENSION (Algorithm 15).

Similarly to the SPY-Method, m-completeness of test suites generated by the SPYH-Method follows from those satisfying the SPY-Condition (Corollary 4.2.2) as for each transition (q, x, y, q') in the reference model, DISTINGUISHFROMSET establishes the convergence of $\upsilon_q.(x/y)$ and $\upsilon_{q'}$ via by satisfying the sufficient condition given in Lemma 4.2.5. In contrast to the SPY-Method, however, the SPYH-Method is not required to always apply the same harmonised state identifiers to separate traces. Instead, it chooses distinguishing traces on-the-fly, opening further possibilities for test suite reduction. As the SPYH-Method employs separation of certain traces to establish $\mathcal{F}(M, m)_{TS}$-convergences via Lemma 4.2.5, its completeness can also be established directly via the S-Condition (Lemma 4.1.5) or the derived abstract H-Condition (Lemma 4.1.7). Fig. 5.7 combines Fig. 5.5 and Fig. 5.6 to visualise how the SPYH-Method employs both dynamic selection of distinguishing traces and dynamic distribution over convergent traces (that is, aspects of both H and SPY-Methods).

Application of the SPYH-Method to running example M_5 (Fig. 5.1) for $m = 4$ exemplifies how this approach results in test suites smaller than those generated by both the H or SPY-Methods. Algorithm 11 first creates state cover V_5 and then via SPYH-DISTINGUISH (Algorithm 9) separates each $\upsilon \in V_5$

Algorithm 13: bestPrefixOfSepSeq(α, β, TS, G)

```
1  π ← minimal trace in CG-lookup(G, α)
2  τ ← minimal trace in CG-lookup(G, β)
3  q_π ← M-after-π
4  q_τ ← M-after-τ
5  minEstimate ← 2|γ| where γ is minimal in Δ_M(q_π, q_τ)
6  bestPrefix ← ϵ
7  if not hasLeaf(π, TS, G) then
8      minEstimate ← minEstimate + |π|
9  if not hasLeaf(τ, TS, G) then
10     minEstimate ← minEstimate + |τ|
11 foreach x ∈ Σ_I, y ∈ Σ_O do
12     if hasExtension(π, x, y, TS, G) then
13         if hasExtension(τ, x, y, TS, G) then
14             if x/y ∈ Δ_M(q_π, q_τ) then return (0, ϵ)
15             if q_π-after-x/y = q_τ-after-x/y then continue
16             (e, γ') ← bestPrefixOfSepSeq(α.(x/y), β.(x/y), TS, G)
17             if e = 0 then return (0, ϵ)
18             if e ≤ minEstimate then
19                 minEstimate ← e; bestPrefix ← (x/y).γ'
20         else
21             e ← estimateGrowthOf(q_π, q_τ, x, y)
22             if e ≠ 1 then
23                 if hasLeaf(π, TS, G) then e ← e + 1
24                 else if not hasLeaf(π.(x/y), TS, G) then
25                     e ← e + |π| + 1
26             if not hasLeaf(τ, TS, G) then e ← e + |τ|
27             if e ≤ minEstimate then
28                 minEstimate ← e; bestPrefix ← (x/y)
29     else
30         e ← estimateGrowthOf(q_π, q_τ, x, y)
31         if e ≠ 1 then
32             if hasLeaf(τ, TS, G) then e ← e + 1
33             else if not hasLeaf(τ.(x/y), TS, G) then
34                 e ← e + |τ| + 1
35         if not hasLeaf(π, TS, G) then e ← e + |π|
36         if e ≤ minEstimate then
37             minEstimate ← e; bestPrefix ← (x/y)
38     return (minEstimate, bestPrefix)
```

Algorithm 14: hasLeaf(α, TS, G)

```
W ← CG-lookup(G,α)
return True iff there exists some α′ ∈ W that is maximal in TS
```

Algorithm 15: hasExtension(α, x, y, TS, G)

```
W ← CG-lookup(G,α)
return True iff there exists an α′ ∈ W such that α′.(x/y) ∈ pref(TS)
```

Algorithm 16: estimateGrowth(q, q', x, y)

```
if x/y ∈ Δ_M(q, q′) then
    return 1
if q-after-x/y = q′-after-x/y ∨ {q-after-x/y, q′-after-x/y} = {q, q′} then
    return 2|Q|          // longer than any minimal γ ∈ Δ_M(q, q′)
return 2|γ| + 1 where γ is minimal in Δ_M(q-after-x/y, q′-after-x/y)
```

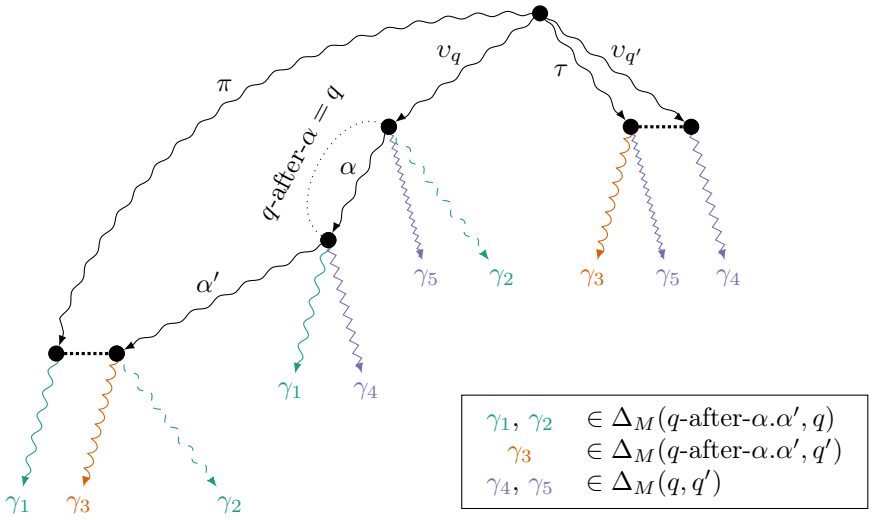


Figure 5.7: Visualisation of the SPYH-Method separating some $\upsilon_q.\alpha.\alpha'$ as required by the S-Condition. Dotted lines between nodes indicate that the traces reaching these nodes are $\mathcal{F}(M,m)_{TS}$-convergent. Traces $\upsilon_q.\alpha$ and υ_q converge but are not yet $\mathcal{F}(M,m)_{TS}$-convergent and hence cannot be distributed over. Note how different traces may be used to distinguish the same pairs of states, such as γ_1 and γ_2 which distinguish the state reached by $\upsilon_q.\alpha.\alpha'$ from q, and how distinguishing traces may be distributed over $\mathcal{F}(M,m)_{TS}$-convergent traces.

from traces in $V_5 \setminus \{\upsilon\}$. In contrast to the SPY-Method, the call to SPYH-DISTINGUISH(υ, V_5, TS, G) here does not simply append fixed harmonised state identifiers after υ, but instead dynamically selects for each $\upsilon' \in V_5 \setminus \{\upsilon\}$ a trace distinguishing the states reached by υ and υ', using heuristic function BEST-PREFIXOFSEPSEQ to check whether the traces to separate (or $\mathcal{F}(M_5, m)_{TS}$-convergent traces thereof) are already extended by prefixes of distinguishing traces. For example, if $(\upsilon_{q_2}, \upsilon_{q_3})$ are to be separated and the test suite already contains $cb/11$ from the state cover V_5, then the SPYH-Method may choose $b/1$ as distinguishing trace, which only extends test case $cb/11$ to $cbb/111 = \upsilon_{q_3}.(b/1)$ with $\upsilon_{q_2}.(b/1) = cb/11 \in pref(cbb/111)$.

Next, transitions verified by V_5 are collected analogously to the SPY-Method and remaining transitions are sorted by length of the state cover traces reaching their source and target states. Thus, the first transition to consider in the loop beginning in line 7 of Algorithm 11 is $(q_0, b, 0, q_0)$, as $|\upsilon_{q_0}| + |\upsilon_{q_0}| = 2|\epsilon| = 0$. In handling an unverified transition, the SPYH-Method again differs from the SPY-Method in the use of SPYH-DISTINGUISH described above. This approach also differs from the H-Method by allowing the distribution of distinguishing traces over convergent traces, possibly resulting in the reduced branching of the test suite described for M_5 and the SPY-Method in the previous subsection.

The resulting test suite for M_5 and $m = 4$ contains 26 IO-traces which contain 15 distinct maximal input sequences.

5.2 Other Test Strategies

Section 5.1 does not provide a full catalogue of all strategies devised for language-equivalence testing on FSMs, as I have only considered strategies that can be applied to arbitrary FSMs[8] and that are able to generate $\mathcal{F}(M, m)$-complete test suites for $m \geq n$. In this section, I provide a short selection of strategies that do not meet these criteria or have not been considered for other reasons.

First, language-equivalence as well as reduction on arbitrary FSMs can be tested by a so-called "brute force" strategy based on product FSMs (described, for example, in [82, Section 4.5]), which enumerates all input sequences up to length $m \cdot n$. I have not considered this strategy in further detail here, as it is not feasible for all but the smallest FSMs and alphabet sizes.

Next, there exist several strategies that rely on the existence of certain structures in or properties of the reference model. For example, the PDS and ADS-Methods described in [104] can be applied only to reference models that have preset or adaptive distinguishing sequences, which essentially requires that for each state in the reference model there exists an input sequence distinguishing it from all other states[9]. As not all FSMs exhibit such sequences for all states, I have not considered these methods.

[8] Recall that it is possible to transform an arbitrary FSM into a language-equivalent minimal observable FSM.

[9] In the case of the PDS-Method, it is even required that a single input sequence exists that distinguishes all states.

Furthermore, there exist several strategies that are m-complete only if $m = n$ or $m \leq n$. These include the UIOv-Method described in [22], the SAT-based approach developed in [86], as well as the P-Method described in [101].

There also exist several strategies that coincide with strategies described in Section 5.1 in the general case of arbitrary FSMs and in the context of equivalence testing. For example, the FF-Method [90] is parameterised by a fault function specifying the kind of faults to test against. This allows for smaller test suites than the H-Method upon which it is based, but if language-equivalence is considered, then both strategies coincide. As another example, the SVS-Method proposed in [104] essentially is a particular implementation of the Wp-Method that employs state verifying sequences as state identification sets, where a sequence $\bar{x}$ verifies state q if it distinguishes q from all other state of the reference model. If for some state no such verifying sequence exists, then a set of distinguishing sequences (i.e. a state identifier) is employed.

Finally, the S-Method presented in [106], a complete strategy for equivalence testing on completely specified DFSMs for arbitrary $m \geq n$, has come to my attention too late to be fully included in the generalisation and formalisation performed in this work. This strategy employs auxiliary data structures in addition to the convergence graph already employed by the SPY and SPYH-Methods. While developing in Section 6.3 a generalisation of the strategies described in the previous section, I derive a partial implementation of the S-Method, denoted $\mathrm{S}_{partial}$-Method, which improves upon the SPYH-Method only in that it does not necessarily establish convergence for all unverified transitions. For some reference models and transitions, the establishment of convergence may incur more extensions of the test suite than are saved by exploiting the convergence in subsequent steps.

Chapter 6

Generic Frameworks for Complete Test Strategies

In this chapter, I develop generic frameworks that are suitable to implement and prove complete the strategies listed in Section 5.1. I here use the term *framework* to denote an abstract implementation of a test suite generation process that provides a high level control flow, leaving the concrete implementation of certain low-level steps to be provided to it as arguments. Concrete strategies are then implemented by supplying suitable implementations of these low level steps to the framework. That is, the frameworks presented here identify and implement the high level steps shared by various strategies but allow low level aspects to differ between strategies.

As an example, consider the W and HSI-Methods (Algorithms 2 and 4). A framework for these two strategies might implement two high level steps: First, an initial test suite is computed as in lines 1 and 3 of both algorithms. Next, the traces in this test suite are extended with sets of traces. The concrete selection of the latter sets of traces then constitutes low level steps that are implemented differently by the two strategies, as the W-Method appends characterisation sets whereas the HSI-Methods employs harmonised state identifiers.

A main objective of this work is to develop frameworks that are extensible and avoid repetition of code and proof steps, as is further detailed in Subsection 6.2.1. Isabelle/HOL supports purely functional programming with higher order functions[1] (see [76]), which are well suited to represent such frameworks. Using so-called *interface lemmata*, the implementation of control flow shared between test strategies as a higher order functions also simplifies the proof effort, as the completeness proof of test suites generated by a framework can be decoupled from the concrete implementations passed to its procedural parameters.

[1] Higher order functions are functions that return a function or exhibit one or more functions as parameters. In the following description, only the latter property is explicitly made use of. Parameters of a higher order function that are themselves functions are called *procedural parameters*.

Together with further benefits, this is discussed in Subsection 6.2.2.

Prior to describing and developing various frameworks, I open this chapter by justifying in Section 6.1 the need to develop such frameworks in the first place. In particular, I consider the question whether any standard implementation of a strategy described in Section 5.1 is already sufficient to simulate all other considered strategies and arrive at a negative result.

Thereafter, Section 6.2 considers objectives in designing generic frameworks and describes in greater detail how higher order functions may be employed in facilitating reuse of code and proofs. This section thus provides a high level overview into the organisation of proofs in the mechanised formalisation described in Part III of this work.

The remaining sections of this chapter introduce three generic frameworks. Section 6.3 introduces the *H-Framework*, which is able to represent all strategies described in Section 5.1 and proves them all complete via the abstract H-Condition (Lemma 4.1.7). Next, Section 6.4 develops a variant of the previous framework that applies to strategies that explicitly create a transition cover as required by the SPY-Condition (Corollary 4.2.2). Thus, this framework does not cover the H-Method, but provides an alternative completeness proof of the remaining strategies. Finally, Section 6.5 introduces a much simpler framework able to cover strategies based on H-Condition (Lemma 4.1.3) that do not exploit convergence. This may also be employed in implementing strategies that closely resemble the H-Method for other conformance relations.

While the first of these frameworks covers all relevant strategies, the remaining frameworks have been added to provide distinct approaches and simpler implementations in exchange for covering fewer strategies.

6.1 Specialisation Relations

The strategies considered in Section 5.1, namely the W, Wp, HSI, H, SPY, and SPYH-Methods, can be grouped by (a) the way in which they select distinguishing traces and by (b) whether they explicitly establish $\mathcal{F}(M,m)_{TS}$-convergences and then distribute distinguishing traces over convergent test cases.

Considering criterion (a), the W, Wp, HSI and SPY-Methods all select distinguishing traces prior to the construction of the test suite in the form of characterisation sets or harmonised state identifiers. That is, they select distinguishing traces in a static way. In contrast, the SPYH and H-Methods select distinguishing traces in a dynamic way depending on the current test suite, allowing the same pair of states to be distinguished by distinct traces in different steps during test suite construction.

Criterion (b) results in a distinct partition, as here the W, Wp, HSI and H-Methods constitute one group due to neither of these strategies exploiting convergences. This group can be further divided by separating the strategies that do establish convergences but do not exploit them from those that do not necessarily establish convergences in the first place. The former subgroup includes the W, Wp, and HSI-Methods, as they contain test suites that can

Table 6.1: Categorisation of the test strategies described in Section 5.1 based on how they select and distribute distinguishing traces.

Strategy	Selection	Distribution
W	static	static (does establish convergences)
Wp	static	static (does establish convergences)
HSI	static	static (does establish convergences)
H	dynamic	static
SPY	static	dynamic
SPYH	dynamic	dynamic

also be generated by certain implementations of the SPY-Method (as will be described later in this section), while the latter subgroup contains only the H-Method. The second group for criterion (b) is composed of the remaining SPY and SPYH-Methods, which not only establish but also exploit convergences by distribution of distinguishing traces over convergent traces.

Table 6.1 summarises this categorisation.

6.1.1 Full Generalisations

In trying to unify the completeness proofs of the considered test strategies, it is a fruitful first step to consider whether any of these strategies constitute merely particular implementations of other strategies, resulting in shared completeness arguments. If every implementation of some strategy Y is a particular implementation of some strategy X, then I call X a *generalisation* of Y. Such generalisation relations hold for several pairs of strategies, based on the following observations, which all assume that the same state cover V is selected by the various methods. Fig. 6.1 visualises the relations discussed in this and subsequent subsections.

First, the Wp-Method (Algorithm 3) can implement the W-Method (Algorithm 2) by selecting the same characterisation set W and for each state q of the reference model selecting $W_q = W$, satisfying the condition required in line 3 of Algorithm 3. This results in applying W after all considered traces, thus coinciding with the W-Method.

Analogously, the HSI-Method (Algorithm 4) is suited to implement the W-Method by selecting the characterisation set W selected in the W-Method as harmonised state identifier for each state q. That is, by selecting $H_q = W$ in line 2 of Algorithm 4, which again results in characterisation set W being applied after all considered traces, coinciding with the W-Method.

Finally, the SPY-Method (Algorithm 7) can recreate the HSI-Method by implementing DISTRIBUTEEXTENSION(α,γ,TS,G) as simply adding $\alpha.\gamma$ to TS. That is, by always selecting α when choosing some trace in $[\alpha]_{TS}$ to append a distinguishing trace after. The test suite computed by this strategy contains the entire test suite computed by the HSI-Method, assuming that both strategies

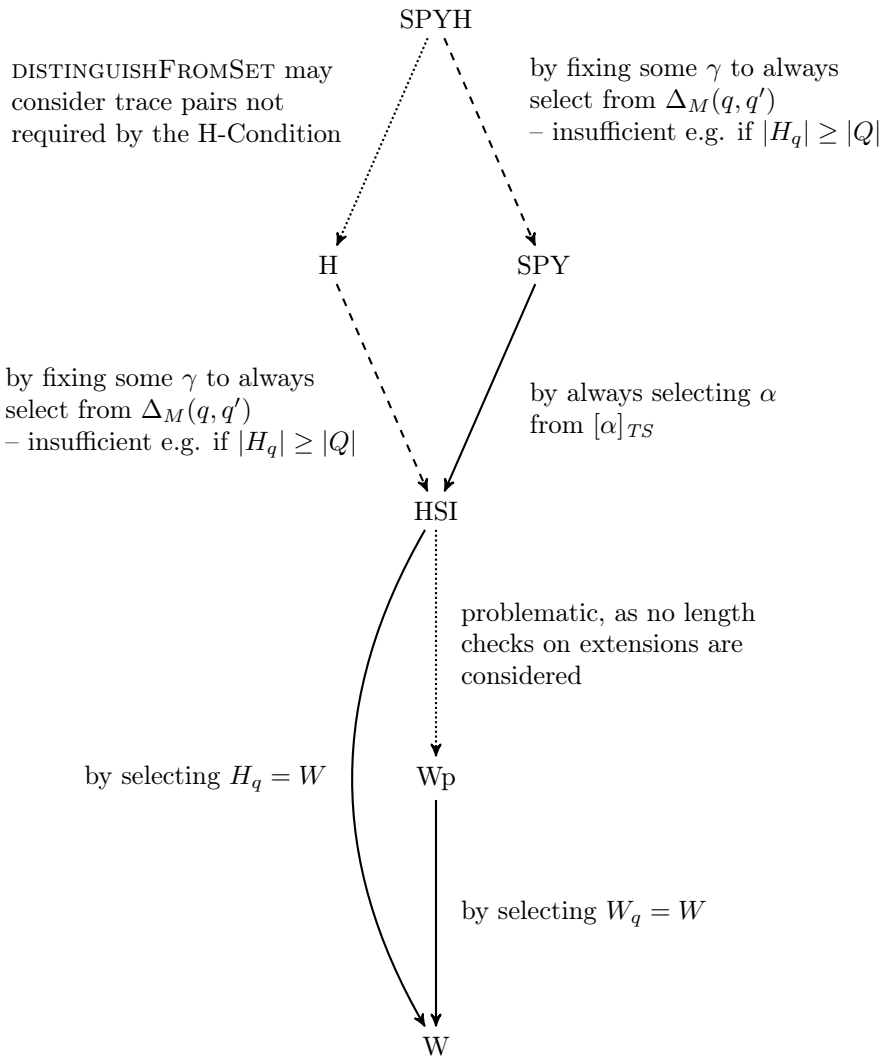


Figure 6.1: Specialisation relations between test strategies. A solid arrow from strategy X to strategy Y indicates that by certain implementation choices, X can generate all test suites generated by Y. A dashed arrow from X to Y indicates that minor modifications are required for X to recreate some but not all implementations of Y. A dotted arrow from X to Y indicates that realising practical implementations of Y via X is not generally possible.

choose the same harmonised state identifiers, by the following argument over Algorithms 4 and 7: Consider some $\alpha \in short_M(V.(\bigcup_{i=0}^{m-n+1}(\Sigma_I \times \Sigma_O)^i))$ handled in line 4 of Algorithm 4. Then there exist $q \in Q$ and ω such that $\alpha = \upsilon_q.\omega$ and $|\omega| \leq m - n + 1$. It can be shown by induction on the length of ω that α is handled identically in the HSI-Method and the SPY-Method with the described concrete implementation of DISTRIBUTEEXTENSION. Let $|\omega| = 0$. Then both strategies handle $\upsilon_q.\omega$ by applying $H_{q\text{-after-}\omega} = H_q$ after υ_q in line 3 of Algorithm 7 and line 6 of Algorithm 4. Thus, suppose that $\omega = (x/y).\omega'$ for some x, y, ω'. If there exists no transition from q with IO-pair x/y in M, then both strategies simply apply x/y after υ_q and ignore ω' in line 15 of Algorithm 7 and line 3 of Algorithm 4. Suppose then that there exists a transition (q, x, y, q') in M. If it is already verified in line 5 of Algorithm 7, then $\upsilon_{q'} = \upsilon_q.(x/y)$ and hence $\alpha = \upsilon_q.(x/y).\omega' = \upsilon_{q'}.\omega'$ where $|\omega'| < |\omega|$. Therefore, both strategies handle α equivalently by the induction hypothesis. Suppose alternatively that (q, x, y, q') is not already verified. Then, if $\omega \in \mathcal{L}_M(q)$, both strategies append $H_{q\text{-after-}\omega}$ after $\upsilon_q.\omega$ in line 9 of Algorithm 7 and line 6 of Algorithm 4. If $\omega \notin \mathcal{L}_M(q)$, then both strategies simply append ω after υ_q in line 12 of Algorithm 7 and line 3 of Algorithm 4, concluding that both strategies handle α equivalently and hence that this implementation of the SPY-Method contains the test suite computed by the HSI-Method. Additionally, this implementation of the SPY-Method does not add test cases not considered in the HSI-Method, as lines 10 and 13 of Algorithm 7, which have not been discussed above, only contribute to adding harmonised state identifiers after shortened extensions of length up to $m - n$ and hence are also covered by the HSI-Method in lines 3 and 6 of Algorithm 4. Thus, the SPY-Method is a generalisation of the HSI-Method, which, together with the Wp-Method, is in turn a generalisation of the W-Method.

6.1.2 Generalisations After Minor Modifications

In addition to the generalisations established in the previous subsection, some strategies can realise at least some implementations of other strategies after minor modifications to the former, often excluding particularly inefficient implementations. I consider a modification to be minor if it only affects heuristics and does not change the types of variables in the algorithms.

For example, the H and SPYH-Methods can simulate implementations of the HSI and SPY-Methods, respectively, if the latter do employ minimal state identifiers, by weakening the decisions as to whether to add distinguishing traces and which distinguishing traces to add.

Consider an implementation of the H-Method (Algorithm 5) and assume that it does not perform the check in line 8 whether the pair of traces to separate is already separable by the test suite[2]. Consider further a symmetric function d such that for any pair q, q' of distinct states in minimal reference model M,

[2]This is, in fact, an equivalent implementation, since if for some (α, β) the condition of line 8 is false in Algorithm 5, the modified implementation may obtain the same result as Algorithm 5 by inserting into the test suite a pair of traces already contained therein.

$d(q, q')$ returns a distinguishing trace of q and q'. That is, suppose that the following holds:

$$\forall q, q' \in Q.\, q \neq q' \longrightarrow d(q, q') = d(q', q) \wedge d(q, q') \in \Delta_M(q, q')$$

Then it constitutes a valid implementation of the modified H-Method to select $d(M\text{-after-}\alpha, M\text{-after-}\beta)$ as separating trace for any pair (α, β) to separate, appending it after both α and β in the test suite. Let

$$H_q = \bigcup_{q' \in Q \setminus \{q\}} d(q, q')$$

denote the set obtained by collecting all distinguishing traces used by d to distinguish $q \in Q$ from other states. By definition of d, this constitutes a state identifier. By symmetry of d, it is also harmonised:

$$q, q' \in Q.\, q \neq q' \longrightarrow d(q, q') \in H_q \cap H_{q'} \cap \Delta_M(q, q')$$

After handling sets A and B (lines 2 and 3 of Algorithm 5), the modified implementation of the H-Method has appended $d(q, q'\text{-after-}\alpha)$ for each pair of distinct states q, q' and trace α of length up to $m - n + 1$ such that all proper prefixes of α are in $\mathcal{L}_M(q')$. That is, harmonised state identifier $H_{q'\text{-after-}\alpha} = H_{M\text{-after-}\upsilon_{q'}.\alpha}$ has been appended after $\upsilon_{q'}.\alpha$ in the test suite, as it would be in the HSI-Method via line 6 of Algorithm 4. Thus, this implementation generates a test suite coinciding with an Implementation of the HSI-Method choosing H_q as harmonised state identifier for each $q \in Q$.

The HSI-Method allows the use of harmonised state identifiers H_q that are not minimal in the sense that they contain more than the at most $|Q|-1$ distinct maximal traces required to distinguish q from the other states in M. In order for H-Method to be able to simulate such cases, the above modification would have to be further extended to allow function d to assign more than a single trace to a pair of states. This would constitute more than a minor modification, and hence I do not consider the H-Method to be a generalisation of the HSI-Method even with minor modifications. Note that the converse also holds, as the HSI-Method would have to be substantially modified to incorporate checks as to whether the test suite is already sufficient to separate certain pairs of traces, as well as the option to not apply the same state identifiers after converging traces.

Analogous to this simulation of the HSI-Method by a modified H-Method, the SPYH-Method could be modified to be able to generate test suites also generated by the SPY-Method. That is, by modifying bestPrefixOfSepSeq to always return $(1, \epsilon)$ and modifying line 4 of SPYH-distinguish (Algorithm 9) to employ a symmetric function d defined as above, it is possible to replicate the distribution of harmonised state identifiers over convergent traces performed by the SPY-Method for minimal harmonised state identifiers. Again analogous to the previous case, any adaptation of the SPYH-Method sufficient to fully cover the SPY-Method for arbitrary state identifiers would require more than minor modifications.

6.1.3 Missing Generalisations

As is visualised in Fig. 6.1, the previous subsections cover full generalisations of the W-Method by the Wp and HSI-Methods as well as the HSI-Method by the SPY-Method, and generalisations after minor modifications of the SPY-Method by the SPYH-Method and the HSI-Method by the H-Method. Furthermore, older strategies are generally not generalisations of more recently developed strategies, as indicated by the missing upwards arrows in Fig. 6.1. For example, the W-Method requires application of the same characterisation set after all considered traces and hence even with minor modifications cannot simulate strategies that append traces with different sets of distinguishing traces depending on the length or reached state of the trace. Similarly, the W, Wp, HSI and SPY-Methods cannot simulate the dynamic choice of distinguishing traces performed by the H and SPYH-Methods, while the H-Method does not exploit convergences.

This leaves open two cases: the relation between the Wp-Method and more recent methods, and the relation between the SPYH and H-Methods. In case of the Wp-Method, any attempt to simulate it via other methods requires distinction whether an extension applied after a trace in the state cover is of length $m - n + 1$. This consideration is not performed by any other test strategy and hence no generalisation is possible using only minor modifications.

Finally, at first glance it may seem possible to simulate the H-Method via the SPYH-Method analogous to the simulation of the HSI-Method by the SPY-Method – that is, by simply forgoing to exploit distribution over convergent traces. Note, however, that function DISTINGUISHFROMSET (Algorithm 12), employed by the SPYH-Method in line 11 of Algorithm 11 to distinguish some traces $\upsilon_q.(x/y)$ and $\upsilon_{q'}$, extends both given traces simultaneously and establishes preservation of divergence between all such extensions. Hence, in line 1 and 4 of recursive calls in Algorithm 12 it may be required to separate a pair $(\upsilon_{q'}.\gamma', \upsilon_q.(x/y).\gamma)$ where γ' is a prefix of γ. Such pairs may not be required to be separated by the H-Condition (Lemma 4.1.3), as this condition only requires preservation of divergence for $V \cup \{\upsilon_q\}.pref((x/y).\gamma)$ when considering extension $(x/y).\gamma$ of υ_q. This obstacle is a direct result of the SPYH-Method using Lemma 4.2.6 to establish convergence of traces, as this lemma requires preservation of convergence for $V \cup \{\upsilon_q.(x/y), \upsilon_{q'}\}.pref(\gamma)$. Thus, strategies that fully apply Lemma 4.2.6 would need to be modified to not do so – or only do so in cases that coincide with the H-Method – in order to simulate the H-Method, which exceeds a minor modification.

6.2 Designing Generic Frameworks

The previous section shows that there exists no minor modification to any of the strategies considered in Section 5.1 that would allow implementing all other strategies via this strategy. At the same time, it is not desirable to develop for each strategy a single independent monolithic algorithm, since this would entail

repetition of not only implementation details but also proof steps to establish completeness.

6.2.1 Objectives

In the search for a preferable approach to implementing and proving complete the strategies, I have considered the following objectives:

1. Repetition of implementation code and proof steps should be minimised.
2. Implementations of the strategies should not be more restrictive than those presented in Section 5.1
3. Implementations and proofs should be extendable in that it should be possible to implement an prove complete some modifications of the strategies without the need to repeat entire implementations or proofs.

As an example for the first objective, consider the W, Wp, and HSI-Methods, which for a given state cover all apply sets of distinguishing traces after the same set of traces. Thus, the computation of this latter set could be extracted into a single algorithm to be employed in implementing these three strategies.

The second objective refers to steps in the strategies that may be implemented in various ways. For example, all strategies compute a state cover but do not prescribe how this computation is to be performed. Thus, implementations of these strategies should not be restricted to a single technique of finding state covers, but instead could, for example, be made configurable as to how a state cover is to be computed.

The third objective calls for approaches that are sufficiently generic to be able to cover at least some modifications of the considered strategies without the need for a complete re-design. Consider the SPYH-Method, which employs the sufficient condition given in Lemma 4.2.5 to establish convergence of a pair of traces. Suppose that a distinct new sufficient condition for establishing convergence is discovered, which leads to a new function to be used for this purpose instead of distinguishFromSet. Then it would be desirable if it were easily possible to prove complete the resulting new strategy obtained by replacing the call to distinguishFromSet in the SPYH-Method with a call to this new function. Naturally, this objective must be limited in scope, as no single practical and efficient approach may cover all possible modifications of complete strategies for language-equivalence testing[3].

[3]For example, there exist complete strategies for testing for quasi-equivalence, a conformance relation coinciding with language-equivalence on completely specified OFSMs, employing a SAT-based approach to FSM-learning (see [86]). These do not follow the approach of strategies presented here, in that they neither explicitly employ a state cover nor iterate over a list of pairs of traces to separate or transitions to verify. That is, implementations of such strategies very likely have few if any aspects in common with the strategies considered here.

6.2.2 Frameworks as Higher Order Functions

In order to satisfy these objectives, I have chosen to develop generic frameworks in the form of higher order functions. These specify a high level structure common to the considered strategies, while implementation details not common to all these strategies are to be provided via the procedural parameters.

This approach offers several benefits. First, the behaviour of a higher order framework – that is, the steps it performs, potentially employing its procedural parameters – is shared between all concrete implementations that provide procedural parameters to it, thus avoiding multiple distinct implementations of steps shared by all strategies. This is instrumental in satisfying the first objective defined above with respect to implementation.

Second, by extracting all behaviour not shared by all strategies into procedural parameters, the behaviour directly fixed by the framework itself is not too restrictive with respect to the second objective.

Third, some modifications to the considered strategies may be implemented by replacing the function passed to some function parameter. Consider a framework F with procedural parameters $P_1, \ldots, P_k$ and a strategy S implemented via F. Then there exists a function representing S that calls $F(I_1, \ldots, I_k)$ for some functions $I_1, \ldots, I_k$. Consider a modification S' of S that affects only a single procedural parameter P_i. In order to implement S' in framework F it may suffice to develop a function I_i' to be passed to F instead of I_i, while not modifying the other parameters. As a concrete example, the frameworks developed in the two subsequent sections indirectly feature the sorting of unverified transitions in the SPYH-Method as a procedural parameter, which allows modifying the sorting strategy of the SPYH-Method without the need to adapt other aspects of the implementation.

Finally, the use of higher order functions as frameworks allows for proofs that establish completeness for all implementations that pass to the procedural parameters functions satisfying certain properties, as opposed to developing separate completeness proofs for each implementation employing the framework. Consider a higher order framework F with procedural parameters P_1, P_2. Instead of unfolding F in the completeness proof of every implementation $I = F(I_1, I_2)$, it may be possible to provide a single proof that F generates complete test suites for all implementations $I = F(I_1, I_2)$ where I_1 and I_2 satisfy some properties ϕ_1 and ϕ_2, respectively. That is, it may be established that the following holds[4]:

$$\forall I_1, I_2.\, \phi_1(I_1) \wedge \phi_2(I_2) \longrightarrow F(I_1, I_2) \text{ is complete for } \mathcal{F}(M, m) \tag{6.1}$$

If this property holds, then an implementation $I = F(I_1, I_2)$ can be proven complete by establishing properties of I_1 and I_2 – that is, without any further reference to framework F.

[4] In the following I omit the passing reference model M and upper bound m to calls to frameworks in order to simplify the presentation. That is, $I = F(I_1, I_2)$ should be written more precisely as $I(M, m) = F(M, m, I_1, I_2)$.

Establishing such implications has several further benefits: First, it allows the reuse of proofs. Suppose that property 6.1 holds and that an implementation $I = F(I_1, I_2)$ has been proven complete by providing proofs of $\phi_1(I_1)$ and $\phi_2(I_2)$. Now consider the task of proving complete an implementation $I' = F(I_1, I_2')$ that shares I_1 with the previous implementation. In this case, the proof of $\phi_1(I_1)$ may be reused, so that it remains only to prove $\phi_2(I_2')$. This reuse of proofs simplifies proving complete new implementations that share some functions with implementations already proven complete, thus also reducing the effort required in establishing completeness of strategies that differ from already handled strategies only in minor aspects such as heuristics.

Next, by independently establishing properties of implementations of procedural parameters, it may be possible to obtain complete test strategies "for free". Suppose that distinct implementations $I = F(I_1, I_2)$ and $I' = F(I_1', I_2')$ have been proven complete by property 6.1. Then, $\phi_1(I_1), \phi_1(I_1'), \phi_2(I_2)$, and $\phi_2(I_2')$ have been established. By property 6.1, this immediately establishes completeness of two further implementations: $F(I_1, I_2')$ and $F(I_1', I_2)$. As a more concrete example, the frameworks developed in subsequent sections are able to derive in this way the completeness of a test strategy that combines the selection of sets of distinguishing traces employed in the Wp-Method with the distribution over convergent traces performed by the SPY-Method by treating both aspects as procedural parameters.

Finally, by establishing completeness of concrete implementations via properties such as 6.1, it is possible to hide implementation details of the functions supplied to the procedural parameters. Consider the following completeness proof of some implementation $F(I_1, I_2)$ via property 6.1 as it might be written in a proof assistant:

```
lemma l1:  φ1(I1)
   proof {...  unfolding I1 ...}
lemma l2:  φ2(I2)
   proof {...  unfolding I2 ...}
lemma l3:  ∀I1, I2. φ1(I1) ∧ φ2(I2) ⟶ F(I1, I2, ) is complete for F(M, m)
   proof {...  unfolding F ...}
lemma l4:  F(I1, I2) is complete for F(M, m)
   proof {
     have φ1(I1) ∧ φ2(I2) ⟶ F(I1, I2, ) is complete for F(M, m)
       from applying the universal elimination rule to lemma l3
     moreover have φ1(I1) from lemma l1
     moreover have φ2(I2) from lemma l2
     ultimately show F(I1, I2) is complete for F(M, m)
       by the conjunction introduction rule and modus ponens }
```

In this script, only the proof of lemma `l1` needs to unfold the definition of I_1.

Thus, if the implementation of I_1 changes, only this single proof needs to be updated, as the proofs of lemmata `l2` to `l4` are not affected. In [123], Woos et al. introduce the concept of *interface lemmata*, which establish properties of functions (here property ϕ_1 for function I_1) and then hide implementation details of these functions in later proofs by employing interface lemmata instead of unfolding the function definitions (here the proof of lemma `l4` uses lemma `l1` instead of unfolding I_1). In conjunction with properties such as 6.1, higher order frameworks are well suited to the introduction of various interface lemmata that describe the desired properties of functions to be passed to the procedural parameters of the framework. Therefore, they improve the maintainability of mechanised formalisations, as changes to a single implementation of some procedural parameter only affect the corresponding interface lemma.

Note here that functions passed to the procedural parameters of a higher order function may themselves be higher order functions. For example, in some implementation $I = F(I_1, I_2)$, function I_1 might merely be a call $F_1(I_3, I_4)$ to some higher order function F_1. This allows establishing properties analogous to 6.1 for proving $\phi_1(F_1(I_3, I_4))$, for example

$$\forall I_3, I_4.\, \phi_3(I_3) \wedge \phi_4(I_4) \longrightarrow \phi_1(F_1(I_3, I_4))$$

With respect to developing frameworks for the test strategies described in Section 5.1, this allows implementing high level structures shared by some but not all of the strategies, where these strategies still differ in certain lower level aspects. For example, the W, Wp, and HSI-Methods all append sets of distinguishing traces after the same set of traces, which allows for a common implementation, but differ in the selection of sets for each trace, which may be extracted into a procedural parameter.

Finally, it is not mandatory that properties such as ϕ_1 in property 6.1 are limited in scope to implementations of a single procedural parameter. In some cases, it may be required to show that implementations of several procedural parameters interact in certain ways. That is, in the frameworks developed in subsequent sections, completeness is often established via properties such as

$$\forall I_1, I_2, I_3.\, \phi_1(I_1) \wedge \phi_2(I_2, I_3) \wedge \phi_3(I_3) \longrightarrow F(I_1, I_2, I_3) \text{ is complete for } \mathcal{F}(M, m)$$

For example, some of the frameworks developed in subsequent sections exhibit separate parameters for separating traces in the state cover from each other and for separating extensions of the state cover from the state cover. For strategies such as the HSI-Method, it is not desirable to require the implementation of the two procedural parameters to establish preservation of divergence in isolation of each other, as this might require the implementation of the second parameter to add again some state identifiers to traces in the state cover. Thus, a property over both implementations may be added that ensures that they employ the same state identifiers (see, for example, Lemma 6.3.1).

6.3 Framework Based on the H-Condition

As described in Section 5.1, all strategies for language-equivalence testing considered in the present work can be proven m-complete using the S-Condition (Lemma 4.1.5), since establishing convergence via Lemma 4.2.6 results in preserving the divergences required by that condition. Thus, these strategies can also be proven m-complete using the abstract H-Condition (Lemma 4.1.7), which weakens requirements on the exact composition of the test suite.

There furthermore exist several intermediate steps of test suite construction that are shared by all considered strategies. First, all strategies begin by selecting a state cover of the reference model. Next, they all extend the traces in the state cover with suitable extensions up to length $m - n + 1$. In the W, Wp, HSI and H-Methods this is performed directly, whereas in the SPY and SPYH-Methods this is performed by extending $\upsilon_q.(x/y)$ for unverified transitions (q, x, y, q') up to length $m - n$. Additionally, for each trace in the state cover and each associated extension, all considered strategies except the H-Method establish preservation of convergence along the trace followed by the extension, combined with the state cover. The strategies differ. however, in how they establish preservation of convergence, ranging from characterisation sets in the W-Method to dynamically appending distinguishing traces in the SPYH-Method.

Note here that the handling of extensions is split into three parts in the SPY and SPYH-Methods: First, they handle the state cover in isolation, which corresponds to handling all extensions of length 0. Next, they handle extensions of υ_q beginning with x/y for each unverified transition (q, x, y, q'). Finally, they handle extensions x/y such that $\upsilon_q.(x/y) \notin \mathcal{L}(M)$. In the following, I apply this split in implementing the W, Wp, HSI, and H-Methods as well, as it covers all required extensions (see Section 6.3.1) and these strategies do not prescribe the order in which extensions are to be considered[5].

The shared intermediate steps constitute a framework for a generic implementation of the considered strategies, which treats the concrete implementations of how these steps are to be performed as procedural parameters. Algorithm 17 presents this framework as a higher order function with the following five procedural parameters:

First, function getStateCover is expected to compute a state cover for the reference model. Its application to the reference model constitutes the first step of Algorithm 17.

Second, function handleStateCover is expected to create a test suite TS that preserves divergence of a given state cover V, as well as a convergence graph G corresponding to this test suite. The returned test suite is furthermore expected to contain the state cover. This function thus performs a part in ensuring satisfaction of the conditions (1.) and (2.) of the abstract H-Condition by establishing that the given state cover V of reference model M is passed by

[5] More precisely, in the W, Wp, and HSI-Methods the order does not affect the generated test suites.

Algorithm 17: H-Framework

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Input : function getStateCover
Input : function handleStateCover
Input : function sortTransitions
Input : function handleUnverifiedTransition
Input : function handleUndefinedIOPair
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
1  V ← getStateCover(M)
2  (TS, G) ← handleStateCover(M, V)
3  U ← {(q, x, y, q') ∈ h_M | υ_q.(x/y) ≠ υ_q'}  // unverified transitions
4  U ← sortTransitions(U,V)
5  foreach t ∈ U do
6      (TS, G) ← handleUnverifiedTransition(M, V, t, m, TS, G)
7  foreach q ∈ Q, x ∈ Σ_I, y ∈ Σ_O such that x/y ∉ L_M(q) do
8      TS ← handleUndefinedIOPair(M, V, q, x, y, TS, G)
9  return TS
```

any $I \in \mathcal{F}(M,m)_{TS}$ and that V is $\mathcal{F}(M,m)_{TS}$-divergence-preserving. Later steps are hence not required to repeatedly check or establish these properties. In Algorithm 17, this function is applied to the state cover computed via getStateCover. Function handleStateCover is inspired by the initial handling of the state cover in the SPY and SPYH-Methods (see lines 2-4 of Algorithm 7 and lines 2-5 of Algorithm 11). For the W, Wp, HSI, and H-Methods, it covers the handling of empty extensions of V.

Next, function sortTransitions is expected to sort the unverified transitions based on a state cover. It is incorporated into the framework to support the SPY and SPYH-Methods. In the remaining strategies, no such sorting is required, but in the case of the H-Method it may impact the generated test suite by controlling the order in which later steps are performed. In Algorithm 17, it is based on the state cover computed via getStateCover.

Thereafter, function handleUnverifiedTransition is expected to handle an unverified transition (q, x, y, q') for a given state cover V by considering for $\upsilon_q.(x/y)$ all extensions γ of length up to $m - n$ such that the proper prefixes γ' of γ satisfy $\upsilon_q.(x/y).\gamma' \in \mathcal{L}(M)$. For each such γ, it is expected that handleUnverifiedTransition updates test suite TS to TS' such that $\{\upsilon_q.(x/y)\}.pref(\gamma)$ is passed by any $I \in \mathcal{F}(M,m)_{TS'}$ and $\{\upsilon_q.(x/y)\}.pref(\gamma)$ is also $\mathcal{F}(M,m)_{TS'}$-divergence-preserving. That is, handleUnverifiedTransition is expected to ensure that conditions (1.) and (2.) of the abstract H-Condition are met for extension $(x/y).\gamma$ of $\upsilon_q \in V$. In Algorithm 17, this function is applied to all unverified transitions. It is designed to support implementation of the handling of unverified transitions in the SPY and SPYH-Methods

(see the loops beginning in line 5 of Algorithm 7 and line 7 of Algorithm 11, respectively), which also allows implementing the handling of non-empty extensions as required by the W, Wp, HSI, and H-Methods.

Finally, function HANDLEUNDEFINEDIOPAIR is expected to update the test suite for a given state cover V such that for a given state $q \in Q$, input $x \in \Sigma_I$, and output $y \in \Sigma_O$, it holds for all $I \in \mathcal{F}(M, m)$ that $\upsilon_q.(x/y)$ is contained in $\mathcal{L}(I)$ if and only if it is contained in $\mathcal{L}(M)$. In Algorithm 17, this function is applied to cover extension $x/y \notin \mathcal{L}_M(q)$ of υ_q.

Beginning with Algorithm 17, I assume that algorithms and their procedural parameters do not modify their parameters, in order to avoid repeating similar assumptions for all subsequent lemmata. For example, HANDLESTATECOVER(M, V) is assumed to not modify M or V. This is equivalent to assuming that arguments passed to function calls are copies of the values stored in the passed variable. In the Isabelle/HOL formalisation described in Part III, this assumption is not required explicitly, as I only employ pure functions (that is, functions without side effects). The mechanised formalisation thus also provides implementations of previously introduced algorithms such as DISTRIBUTEEXTENSION (Algorithm 1) that explicitly return any modifications of inputs as additional outputs.

The following lemma formalises these expectations and shows that any combination of implementations of these functions satisfying the assumptions results in a strategy for generating complete test suites.

Lemma 6.3.1. *Let $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ be an FSM with with $|Q| = n \leq m$. Suppose that the following properties hold over the arguments passed to the procedural parameters of Algorithm 17:*

ϕ_1^{H}) GETSTATECOVER(M) *returns a state cover of M.*

ϕ_2^{H}) *For all TS, G, V, if* HANDLESTATECOVER(M, V) *returns (TS, G), then it holds that*

- *(a) V is $\mathcal{F}(M, m)_{TS}$-divergence-preserving,*
- *(b) $\mathcal{F}(M, m)_{TS} \subseteq \mathcal{F}(M, m)_V$, and*
- *(c) G is a valid convergence graph for M and $\mathcal{F}(M, m)_{TS}$.*

ϕ_3^{H}) *For all U, V,* SORTTRANSITIONS(U, V) *returns the same set of transitions as U. That is,* SORTTRANSITIONS(U, V) *may only change the order in which the elements of U are to be handled in the loop in line 5.*

ϕ_4^{H}) *For all TS, TS', G, G', transitions $t = (q, x, y, q') \in h_M$, and state cover V of M, if*

- HANDLEUNVERIFIEDTRANSITION(M, V, t, m, TS, G) *returns the pair (TS', G'),*
- *the test suite computed by* HANDLESTATECOVER(M, V) *is contained in TS,*

- *V is $\mathcal{F}(M,m)_{TS}$-divergence-preserving,*
- *$\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_V$, and*
- *G is a valid convergence graph for M and $\mathcal{F}(M,m)_{TS}$,*

then all of the following holds:

(a) *$pref(TS) \subseteq pref(TS')$,*

(b) *for each $\gamma \in (\Sigma_I \times \Sigma_O)^*$ with $pref((x/y).\gamma) \setminus \{(x/y).\gamma\} \subseteq \mathcal{L}_M(q)$ and $|\gamma| \leq m - n$ it holds that*

i. *$V \cup \{v_q.(x/y)\}.pref(\gamma)$ is $\mathcal{F}(M,m)_{TS'}$-divergence-preserving, and*

ii. *$\mathcal{F}(M,m)_{TS'} \subseteq \mathcal{F}(M,m)_{V \cup \{v_q.(x/y)\}.pref(\gamma)}$,*

(c) *G' is a valid convergence graph for M and $\mathcal{F}(M,m)_{TS'}$.*

ϕ_5^{H}) *For all TS, TS', G, state covers V of M and $q \in Q, x \in \Sigma_I, y \in \Sigma_O$, it holds that if* handleUndefinedIOPair(M, V, q, x, y, TS, G) *returns TS' and G is a valid convergence graph for M and $\mathcal{F}(M,m)_{TS}$, then*

- *$pref(TS) \subseteq pref(TS')$, and*
- *$\mathcal{F}(M,m)_{TS'} \subseteq \mathcal{F}(M,m)_{\{v_q.(x/y)\}}$.*

Then, applying values M, m and functions getStateCover, handleStateCover, sortTransitions, handleUnverifiedTransition, *and* handleUndefinedIOPair *to* H-Framework *results in a test suite TS such that its prefix closure $pref(TS)$ is m-complete for testing language-equivalence against reference model M.* ⊣

Proof. I establish this lemma by showing that under the given assumptions, the obtained test suite satisfies the abstract H-Condition introduced in Lemma 4.1.7. Furthermore, if some $I \in \mathcal{F}(M,m)$ does not pass the returned test suite with respect to M, then it is by definition not language-equivalent to M. Hence, it is sufficient to show exhaustiveness of the returned test suite. By ϕ_1^{H}, line 1 of Algorithm 17 assigns to V a state cover of M. Let $I \in \mathcal{F}(M,m)$. Therefore, it is sufficient to show that, assuming I passes the returned test suite, for all $q \in Q$ and $\gamma \in X_q$ all of the following hold:

1. I passes $V \cup \{v_q\}.pref(\gamma)$, that is
$$\mathcal{L}(I) \cap (V \cup \{v_q\}.pref(\gamma)) = \mathcal{L}(M) \cap (V \cup \{v_q\}.pref(\gamma))$$
2. $V \cup \{v_q\}.pref(\gamma)$ is $\{I\}$-divergence-preserving. That is, traces $\alpha, \beta \in V \cup \{v_q\}.pref(\gamma)$ that diverge in M also diverge in I.

where X_q is the set is the set of relevant extensions of v_q, defined as follows

$$X_q := \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$$

Let TS_i and G_i denote the values of $pref(TS)$ and G *after* line i of Algorithm 17 has been performed for the last time. For example, TS_2 is the prefix

closure of the initial test suite created by HANDLESTATECOVER, whereas G_5 is the value of G after HANDLEUNVERIFIEDTRANSITION has been applied to all $t \in U_4$. As line 9 performs no actions other than returning TS without further modification, $TS_9 = TS_7$ and $G_9 = G_7$ hold.

Before deriving (1.) and (2.), I establish some intermediate properties of Algorithm 17. First, by ϕ_3^{H}, lines 3 and 4 assign the same set of transitions to U, namely the set of all transitions not covered by the state cover V.

Next, by ϕ_2^{H} the call to HANDLESTATECOVER in line 2 returns a valid convergence graph G_2 and a test suite TS_2. By assumption ϕ_4^{H} and induction over the size of U, which is finite by construction and ϕ_3^{H}, it holds for each individual call to function HANDLEUNVERIFIEDTRANSITION in line 6 that the test suite passed to it contains the test suite TS_2 computed by HANDLESTATECOVER(M, V) and that the graph passed to it is a valid convergence graph (G_2 in the base case, the valid graph obtained by the previous call in the induction step). Thus, by ϕ_4^{H}, G_5 is valid and the resulting test suite TS_5 contains TS_2 as well as the test suites returned by all iterations of line 6. That is, TS_5 contains the combined result of applying HANDLEUNVERIFIEDTRANSITION to all $t \in U$.

Analogously, using ϕ_5^{H} and the validity of G_5, each graph passed to HANDLEUNDEFINEDIOPAIR in line 8 is valid and hence the final test suite $TS_9 = TS_7$ contains TS_5 as well as all test suites obtained by the calls to HANDLEUNDEFINEDIOPAIR.

I establish Satisfaction of (1.) and (2.) for all $q \in Q$ and $\gamma \in X_q$ by induction on the length of γ. Note that by construction of X_q, there exist α and x/y such that $\gamma = \alpha.(x/y)$, $\alpha \in \mathcal{L}_M(q)$, $|\alpha| \leq m - n$, and $x/y \in \Sigma_I \times \Sigma_O$.

Base case: Let $\alpha = \epsilon$ and hence $\gamma = x/y$. If $x/y \notin \mathcal{L}_M(q)$, then x/y is considered at some point during execution of the loop in line 7 and handled via HANDLEUNDEFINEDIOPAIR in line 8. As described above, the test suite obtained from this call is contained in the returned TS_9 and hence, by ϕ_5^{H}, as I passes TS_9, then it also passes $\{\upsilon_q.(x/y)\}$. By ϕ_2^{H}, I also passes V and hence satisfies (1.). Since $x/y \notin \mathcal{L}_M(q)$, $V \cup \{\upsilon_q\}.pref(\gamma)$ is $\{I\}$-divergence-preserving if V is $\{I\}$-divergence-preserving. The latter is established by ϕ_2^{H} and the fact that I passing TS_9 implies I passing TS_2. Thus, (2.) also holds.

Suppose next that $x/y \in \mathcal{L}_M(q)$. Then there must exist some transition (q, x, y, q') in M, which may be already covered by V. If $\upsilon_q.(x/y) = \upsilon_{q'}$, then $\upsilon_q.(x/y)$ is contained in V and (1.) and (2.) are already established by HANDLESTATECOVER(M, V) in line 2, following from ϕ_2^{H}. Suppose next that $\upsilon_q.(x/y) \neq \upsilon_{q'}$. In this case, $(q, x, y, q') \in U$ holds and hence (q, x, y, q') is considered in the loop in line 5. By property (b) of ϕ_4^{H} for the case of the empty extension ϵ, I passes $V \cup \{\upsilon_q.(x/y)\}.pref(\epsilon) = V \cup \{\upsilon_q\}.pref(x/y)$ and the latter set is also $\{I\}$-divergence-preserving, which establishes (1.) and (2.).

Induction step: Let $\alpha = (x'/y').\alpha'$ and hence $\gamma = (x'/y').\alpha'.(x/y)$. Since $\alpha \in \mathcal{L}_M(q)$, there must again exist some transition (q, x', y', q') in M. If $\upsilon_q.(x'/y') = \upsilon_{q'}$, then $\upsilon_q.(x'/y').\alpha'.(x/y) = \upsilon_{q'}.\alpha'.(x/y)$ and hence $\alpha'.(x/y) \in X_{q'}$. As $\alpha'.(x/y)$ is shorter than $(x'/y').\alpha'.(x/y)$, (1.) and (2.) hold for q' and $\alpha'.(x/y)$ by the induction hypothesis. Furthermore, from $\upsilon_q.(x'/y') = \upsilon_{q'}$ it follows that $\upsilon_q.(x'/y') \in V$ and hence $V \cup \{\upsilon_q\}.pref((x'/y').\alpha'.(x/y)) =$

$V \cup \{\upsilon_q.(x'/y')\}.pref(\alpha'.(x/y))$. Thus, (1.) and (2.) hold for q and γ.

Suppose next that $\upsilon_q.(x'/y') \neq \upsilon_{q'}$. In this case, (q, x', y', q') is contained in U and hence HANDLEUNVERIFIEDTRANSITION is applied on q, V, (q, x', y', q'), and m, and the resulting test suite is passed by I. Furthermore, $\alpha \in \mathcal{L}_M(q)$ implies $\alpha' \in \mathcal{L}_M(q\text{-after-}x'/y')$. Since $|\alpha'.(x/y)| \leq m - n$, by ϕ_4^{H}, (1.) and (2.) therefore hold for q and $\gamma = (x'/y').\alpha'.(x/y)$. □

I have formalised the above proof in theory file `H_Framework.thy`, discussed in Section 10.1. An implementation of Algorithm 17 in Isabelle/HOL is presented in Appendix E.

Interpreting conditions ϕ_1^{H} to ϕ_5^{H} as predicates over the procedural parameters, Lemma 6.3.1 constitutes the property

$$\begin{aligned}\forall I_1, I_2, I_3, I_4, I_5.\, \phi_1^{\mathrm{H}}(I_1) \wedge \phi_2^{\mathrm{H}}(I_2) \wedge \phi_3^{\mathrm{H}}(I_3) \wedge \phi_4^{\mathrm{H}}(I_2, I_4) \wedge \phi_5^{\mathrm{H}}(I_5) \\ \longrightarrow \text{H-FRAMEWORK}(M, m, I_1, I_2, I_3, I_4, I_5) \\ \text{is complete for } \mathcal{F}(M, m) \text{ w.r.t. } M\end{aligned}$$

Thus, as described in Subsection 6.2.2, proving completeness of implementations using framework H-FRAMEWORK subsequently reduces to establishing properties of the functions passed to the procedural parameters.

Having presented a generic framework for implementing various test strategies, it remains to be shown that it is possible to implement in it the strategies selected in Section 5.1. This requires concrete implementations of the five functional parameters of H-FRAMEWORK (Algorithm 17): GETSTATECOVER, HANDLESTATECOVER, SORTTRANSITIONS, HANDLEUNVERIFIEDTRANSITION, and HANDLEUNDEFINEDIOPAIR. These I provide in the remainder of this section. Implementations in Isabelle/HOL as well as mechanised proofs that the provided implementations do satisfy the corresponding conditions of Lemma 6.3.1 are discussed in Part III.

6.3.1 GETSTATECOVER

State covers can be constructed by a simple reachability analysis of the reference model, as each reachable state must by definition exhibit at least one trace in $\mathcal{L}(M)$ reaching it. Thus, a state cover may be computed by a simple breadth-first search as presented in Algorithm 18. An implementation in Isabelle/HOL is given in Section 8.3, which also proves that this algorithm computes a valid state cover and hence satisfies condition ϕ_1^{H} of Lemma 6.3.1.

6.3.2 HANDLESTATECOVER

In the handling of state covers – that is, in handling extensions of length 0 – none of the considered strategies do exploit distribution of distinguishing traces over convergent traces, as no convergences have been established at that point in the test suite computation. Instead, they all simply add the entire state cover to the test suite and then establish preservation of divergence by appending suitable

Algorithm 18: getStateCoverByBFS($M = (Q, q_0, \Sigma_I, \Sigma_O, h_M)$)

```
1  V ← {(q_0, ε)}                    // map from states to reaching traces
2  X ← {q_0}                         // set of states to consider successors of
3  while X ≠ ∅ do
4      X' ← ∅                        // states to consider in next iteration
5      foreach (q, x, y, q') ∈ h_M such that q ∈ X ∧ q' ∉ keys of V do
6          υ_q ← lookup q in V
7          update V with (q', υ_q.(x/y))
8          X' ← X' ∪ {q'}
9      X ← X'
10 return keys of V
```

distinguishing traces. Considering Table 6.1, this leaves as primary distinguishing feature of implementations of handleStateCover the question whether distinguishing traces are selected dynamically or not. Therefore, I provide two higher order functions as implementations, which can be adapted to cover all considered strategies by again providing suitable functions as arguments.

First, the dynamic choice of distinguishing traces can be realised via repeated application of function SPYH-distinguish (Algorithm 9) as implemented in handleStateCover-Dynamic (Algorithm 19). This constitutes a slight optimisation to the way in which the SPYH-Method applies SPYH-distinguish (see lines 2-5 of Algorithm 11) by incrementally building the set of traces to distinguish some υ_q from, instead of directly passing the entire state cover V, thus avoiding handling both $(\upsilon_q, \upsilon_{q'})$ and $(\upsilon_{q'}, \upsilon_q)$ separately. By this iterative process, the inclusion of V into the test suite in line 1, and the assumptions on initialisation and insertion operations on convergence graphs preserving their validity (see Section 3.3), handleStateCover-Dynamic satisfies condition ϕ_2^{H} of Lemma 6.3.1. The function may also be employed to implement the H-Method, as it performs no merge operations on its convergence graph.

The handling of state covers by strategies that append pre-computed sets of distinguishing traces may be implemented via handleStateCover-Static (Algorithm 20). This function applies after each υ_q in the provided state cover the set of distinguishing traces obtained via parameter getDistinguishingSet.

In order to represent the HSI and SPY-Methods, a function computing harmonised state identifiers such as getHSI (Algorithm 22) might be passed as getDistinguishingSet. The W and Wp-Methods can be realised by passing a function such as getCharSet (Algorithm 23) that computes a characterisation set[6]. Both functions rely in turn on a function getShortestDistTrace (Algorithm 21) that computes a distinguishing trace for a given pair of distinct states and is symmetric with respect to that pair.

[6]Recall that the Wp-Method appends state identifiers instead of the full characterisation set only after extensions of length $m - n + 1$ for $n \leq m$. In handling state covers it hence coincides with the W-Method.

Algorithm 19: handleStateCover-Dynamic

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : state cover V of M
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ and convergence graph G

```
1 TS ← V
2 G ← CG-initialise(TS)
3 X ← ∅                          // divergence-preserving subset of V
4 foreach q ∈ Q do
5 |   SPYH-distinguish(υ_q, X, TS, G)
6 |   X ← X ∪ {υ_q}
7 return (TS, G)
```

Algorithm 20: handleStateCover-Static

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : state cover V of M
Input : function getDistinguishingSet
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$ and convergence graph G

```
1 TS ← V
2 foreach q ∈ Q do
3 |   TS ← TS ∪ {υ_q}.getDistinguishingSet(M, q)
4 G ← CG-initialise(TS)
5 return (TS, G)
```

Algorithm 21: getShortestDistTrace(M, q, q')

```
1 return the shortest and lexicographically smallest trace in Δ_M(q, q')
```

Algorithm 22: getHSI$(M = (Q, q_0, \Sigma_I, \Sigma_I, h), q)$

```
1 return ⋃_{q' ∈ Q \ {q}} getShortestDistTrace(M, q, q')
```

Algorithm 23: getCharSet$(M = (Q, q_0, \Sigma_I, \Sigma_I, h), q)$

```
1 return {getShortestDistTrace(M, q, q') | q, q' ∈ Q ∧ q ≠ q'}
```

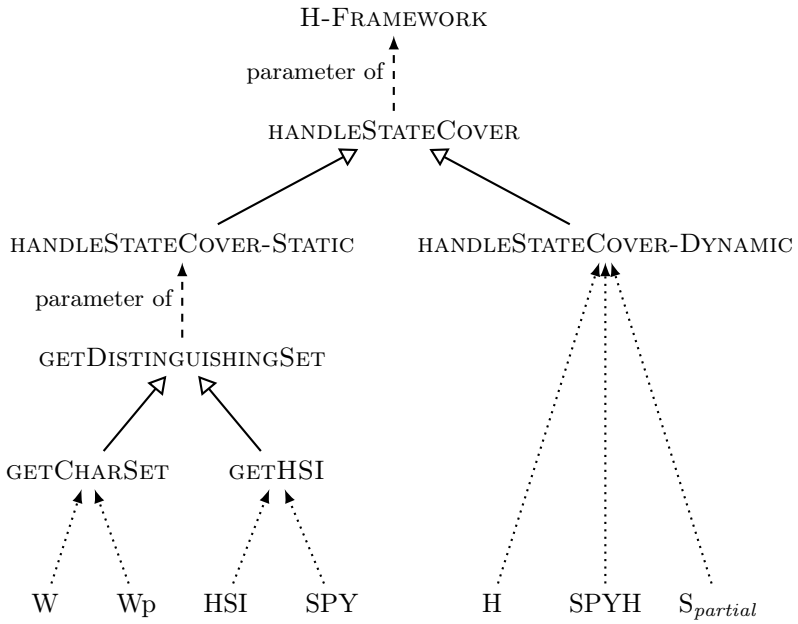


Figure 6.2: Overview of how various strategies may implement parameter handleStateCover of H-Framework (Algorithm 17). An arrow from A to B with a white triangle head indicates that B is a possible implementation of A. Dotted arrows indicate which implementations are employed by each strategy.

handleStateCover-Static satisfies condition ϕ_2^{H} of Lemma 6.3.1 if the sets obtained for each $q \in Q$ via getDistinguishingSet constitute harmonised state identifiers, which holds for both getHSI and getCharSet by construction. Note that in the latter two functions the computation of distinguishing traces could be extracted again as a procedural parameter.

Figure 6.2 visualises how the different strategies implement handleStateCover. The $S_{partial}$-Method is introduced in Subsection 6.3.4, discussing implementations for handleUnverifiedTransition, and does differ from the SPYH-Method only in that procedural parameter of the H-Framework.

6.3.3 sortTransitions

Of the strategies considered in Section 5.1, only the SPYH-Method explicitly sorts unverified transitions, using Algorithm 24. The other strategies apply no sorting, which may be represented by an algorithm that immediately returns unchanged the transition set passed to it[7]. Both strategies trivially satisfy condition ϕ_3^{H} of Lemma 6.3.1, as they do not add or remove any transitions.

Algorithm 24: SPYH-sortTransitions(U, V)

```
1 U' ← sort unverified transitions in ascending order by weight function
    (q, x, y, q') ↦ |υ_q| + |υ_q'|
2 return U'
```

6.3.4 handleUnverifiedTransition

In contrast to the handling of state covers, the considered strategies all differ in the handling of unverified transitions – that is, in the handling of non-empty extensions whose first IO-pair is in the language of the state whose reaching trace they extend. All criteria of distinguishing the strategies presented in table 6.1 are to be considered in deriving concrete implementations of handleUnverifiedTransition.

Analogous to the handling of state covers, I first provide intermediate implementations for dynamic and static selection of distinguishing traces. Strategies with static selection (W, Wp, HSI, SPY) differ in two aspects: First, by their choices of sets of distinguishing traces, which entails the W-Method using characterisation sets, the HSI and SPY-Methods employing harmonised state identifiers, and the Wp-Method combining both approaches depending on the length of the extension after which to append the set. Second, by whether they distribute extensions over converging traces, as only the SPY-Method does.

Algorithm 25 provides an intermediate higher order function handleUT-Static for strategies with static selection. It closely follows the SPY-Method

[7] As they do not prescribe specific orderings, the other strategies could also be implemented using Algorithm 24.

(see lines 6-14 of Algorithm 7) and is configurable in two ways to cover the differentiating aspects of the W, Wp, HSI, and SPY-Methods: First, it takes as additional argument a function getDistSetForLength, which analogously to parameter getDistinguishingSet of handleStateCover-Static (Algorithm 20) is to provide sets of distinguishing traces to append. This parameter differs from getDistinguishingSet by additional parameters for $m - n$ and the length of the extension to append the distinguishing traces after, in order to facilitate implementing the Wp-Method. The second aspect, distribution over convergent traces, may be configured indirectly by selecting a suitable implementation of convergence graph G. For example, a convergence graph that returns only $\{\alpha\}$ for each lookup of $[\alpha]_{TS}$ reduces calls to distributeExtension(α, β, TS, G) to simply appending β after α in the test suite. This, in turn, reduces lines 4 to 6 of Algorithm 25 to simply appending the obtained set of traces after both $v_q.(x/y)$ and $v_{q'}$. It has been explained in Subsection 6.1.1 how this reduces to applying the set of traces as in the HSI-Method, which is identical in this regard to the behaviour of the W and Wp-Methods. In the following, I denote a convergence graph that implements lookup operations in this way as Empty. In contrast, I denote convergence graphs whose lookup implementations depend on previous insertions and mergings as Standard.

Algorithm 25: handleUT-Static

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ with $|Q| = n$
Input : transition $(q, x, y, q) \in h_M$
Input : integer m
Input : test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$
Input : convergence graph G
Input : function getDistSetForLength
Output: test suite $TS' \subseteq (\Sigma_I \times \Sigma_O)^*$ and convergence graph G'

```
1  l ← m − n
2  foreach ω ∈ ⋃_{i=0}^{l} (Σ_I × Σ_O)^i such that pref(ω) \ {ω} ⊆ L_M(q') do
3      if ω ∈ L_M(q') then
4          foreach γ ∈ getDistSetForLength(M, q'-after-ω, l, |ω|) do
5              distributeExtension(M, v_q, (x/y).ω.γ, TS, G)
6              distributeExtension(M, v_q', ω.γ, TS, G)
7      else
8          distributeExtension(M, v_q, (x/y).ω, TS, G)
9          distributeExtension(M, v_q', ω, TS, G)
10 CG-merge(G, v_q.(x/y), v_q')
11 return (TS, G)
```

In a slight abuse of notation, I reuse functions getCharSet and getHSI (Algorithms 23 and 22) as implementations of getDistSetForLength, ignoring the additional parameters. For the Wp-Method, an implementation is

required that returns either the characterisation set already used to cover the state cover (getCharSet) or a state identifier that is a subset of the characterisation set (getHSI), the latter being returned only if the length k of the considered extension is maximal (l). Algorithm 26 provides an implementation in function getCharSetOrHSI.

Algorithm 26: getCharSetOrHSI(M, q, l, k)

```
1 if k = l then
2 |   return getHSI(M, q)
3 else
4 |   return getCharSet(M, q)
```

As visualised in the left-hand side of Fig. 6.3, the handling of unverified transitions in the W, Wp, HSI and SPY-Methods can be implemented by suitable configurations of handleUT-Static. These satisfy condition ϕ_4^{H}(a) of Lemma 6.3.1, since all of the discussed functions only add test cases to the test suite and never remove or shorten already contained test cases. Furthermore, subcondition (i) of condition ϕ_4^{H}(b) is ensured by the appending of suitable harmonised state identifiers[8] after the extensions collected in line 2 of Algorithm 25. Since each such extension is either appended itself or a convergent trace of it is already contained in the test suite, subcondition (ii) is also satisfied. Finally, the assumptions on preservation of graph validity presented in Subsection 3.3 ensure that the graph obtained after finishing the loop remains valid. Additionally, the final graph obtained by the merge operation in line 10 of Algorithm 25 remains valid, as satisfaction of conditions ϕ_4^{H}(a) and ϕ_4^{H}(b) by Lemma 4.2.6 implies that $\upsilon_q.(x/y)$ and $\upsilon_{q'}$ are $\mathcal{F}(M, m)_{TS}$-convergent for value TS of the test suite in line 10. Thus, ϕ_4^{H}(c) also holds. Together with above results for ϕ_4^{H}(a) and ϕ_4^{H}(b), this establishes that handleUT-Static satisfies condition ϕ_4^{H} of Lemma 6.3.1.

A schematic overview of this proof with references to the formalisation is given in Fig. 11.2 in Chapter 10.

The remaining test strategies – H and SPYH-Method – handle an unverified transition (q, x, y, q') in distinct ways. The H-Method (Algorithm 5) does not establish convergence of $\upsilon_q.(x/y)$ with $\upsilon_{q'}$, as it only extends $\upsilon_q.(x/y)$ with suitable extensions up to length $m - n$. In contrast, the SPYH-Method (Algorithm 11) does establish convergence, which, by the sufficient condition given in Lemma 4.2.5, requires extending not only $\upsilon_q.(x/y)$ but also $\upsilon_{q'}$ and preserving divergence between these extensions of both traces. The SPYH-Method employs function distinguishFromSet to perform these steps, while the H-Method as presented in Algorithm 5 handles unverified transitions implicitly as parts of

[8] Recall that getCharSetOrHSI and getCharSet contain the harmonised state identifier computed by getHSI and that these have also been applied after the traces in the state cover by the assumption of condition ϕ_4^{H} if the same functions are supplied to handleStateCover-Static.

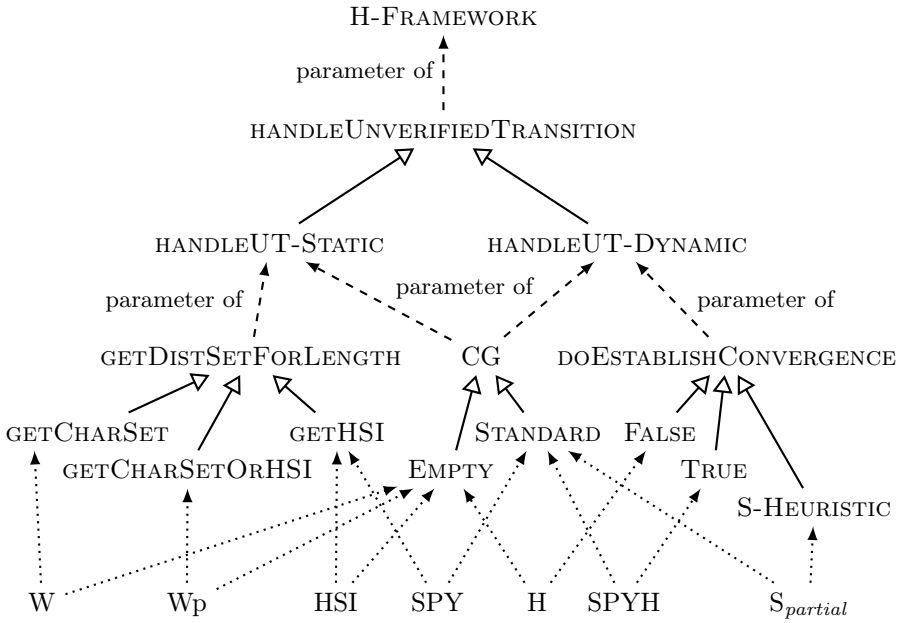


Figure 6.3: Overview of how various strategies may implement parameter handleUnverifiedTransition of H-Framework (Algorithm 17). An arrow from A to B with a white triangle head indicates that B is a possible implementation of A. Dotted arrows indicate which implementations are employed by each strategy. CG abbreviates the implementation of convergence graphs.

sets A, B, C to separate. Thus, a function for explicitly handling an unverified transition according to the H-Method is required. In order to reuse existing functions, which also allows reuse of proofs in the formalisation, I have chosen to employ distinguishFromSet (Algorithm 12) for this purpose. Consider a call of the form

$$\text{distinguishFromSet}(\upsilon_q.(x/y), \upsilon_q.(x/y), V, V, TS, G, m-n)$$

and suppose that convergence graph G is Empty as used above for the W, Wp, and HSI-Methods. By providing $\upsilon_q.(x/y)$ as both the first and second argument and following the implementation given in Algorithm 12, this call performs twice the extensions of $\upsilon_q.(x/y)$ and their preservation of divergence as required by the H-Method. Since the duplicate calls to SPYH-distinguish and distributeExtension in lines 4 and 11 of Algorithm 12 do not modify the test suites obtained by lines 1 and 10, respectively, this approach does furthermore not consider any extensions not required by the H-Method. In the Isabelle/HOL implementation of Algorithm 12, use of the same trace as first and second argument skips the aforementioned superfluous repetitions.

Thus, it is possible to implement handleUnverifiedTransition for the

SPYH-Method by applying DISTINGUISHFROMSET as in the SPYH-Method, while an implementation for the H-Method is obtained by calls to DISTINGUISHFROMSET as previously described for an EMPTY convergence graph. Both approaches satisfy condition ϕ_4^{H} of Lemma 6.3.1 by dynamically appending distinguishing traces to separate extensions from their prefixes and the state cover as required, while also adding the extensions or convergent traces thereof. In the case of the SPYH-Method, $v_q.(x/y)$ and $v_{q'}$ may finally be merged in the convergence graph, whose validity is preserved by the assumptions on merging given in Subsection 3.3 and the satisfaction of the sufficient condition for convergence given in Lemma 4.2.5[9].

Condition ϕ_4^{H} is therefore also satisfied by an implementation that decides for each individual unverified transition (q, x, y, q') whether to handle it as in the H-Method or as in the SPYH-Method. The latter requires larger sets of traces to be divergence-preserving than the former, but in contrast to the former additionally establishes convergence of $v_q.(x/y)$ and $v_{q'}$ in order to increase options for distribution over convergent traces. This approach of dynamically choosing whether to establish convergence for the transition is one of the main contributions of the S-Method introduced by Soucha in [106]. As briefly discussed in Section 5.2, I have not fully considered this strategy in this work. I have, however, incorporated the possibility of dynamically deciding whether to establish convergence for a given transition into a higher order function HANDLEUT-DYNAMIC implemented in Algorithm 27. This function applies DISTINGUISHFROMSET either as in the SPYH-Method (lines 4 and 5) or as suitable for the H-Method (line 7). The decision between these approaches is performed using a function DOESTABLISHCONVERGENCE passed as additional argument to HANDLEUT-DYNAMIC. By implementing DOESTABLISHCONVERGENCE as a function FALSE that just returns '`False`', convergence is never explicitly established, thus realising the handling of transitions in the H-Method. Conversely, by providing a function TRUE that always returns '`True`', HANDLEUT-DYNAMIC behaves in accordance with the SPYH-Method. Finally, Algorithm 28 generalises the heuristic of the S-Method as implemented in Algorithm 25 of [106]. This heuristic chooses to establish convergence for some transition (q, x, y, q') if there exists a still unverified transition such that in handling that transition the convergence of $v_q.(x/y)$ and $v_{q'}$ may extend the options available for distribution over convergent traces. More precisely, convergence is established for (q, x, y, q') if there exists an unverified transition whose source state is reachable from q' within at most $m - n$ transitions. I denote the partial implementation of the S-Method obtained by this heuristic as $\mathrm{S}_{partial}$-Method.

The right-hand side of Fig. 6.3 visualises how the H, SPYH, and $\mathrm{S}_{partial}$-Methods implement HANDLEUNVERIFIEDTRANSITION. In all other functional parameters supplied to H-FRAMEWORK, the SPYH and $\mathrm{S}_{partial}$-Methods coincide, as already indicated in Fig. 6.2. The $\mathrm{S}_{partial}$-Method differs from the S-Method as implemented in [106] for example in selecting a divergence preserv-

[9] Employing this step also in implementing the H-Method has no effect, as lookup operations on EMPTY are not affected by merging.

Algorithm 27: HANDLEUT-DYNAMIC

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : transition $(q, x, y, q') \in h_M$
Input : integer m
Input : test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$
Input : convergence graph G
Input : function DOESTABLISHCONVERGENCE
Output: updated test suite TS and convergence graph G

```
1 l ← m − n
2 DISTRIBUTEEXTENSION(υ_q, (x/y), TS, G)
3 if DOESTABLISHCONVERGENCE(M, (q, x, y, q'), l, TS) then
4 |   DISTINGUISHFROMSET(υ_q.(x/y), υ_q', V, V, TS, G, l)
5 |   CG-MERGE(G, υ_q.(x/y), υ_q')
6 else
7 |   DISTINGUISHFROMSET(υ_q.(x/y), υ_q.(x/y), V, V, TS, G, l)
8 return (TS, G)
```

Algorithm 28: S-HEURISTIC$(M, (q, x, y, q'), l, TS)$

return `True` iff there exists an unverified transition (q'', x', y', q''') of M such that there exists a path from q' to q'' of length at most l

ing state cover and in not employing state domains, a data structure introduced in [106] for keeping track of divergence between convergent classes. See Chapter 11 of [106] for further contributions introduced with the S-Method.

6.3.5 HANDLEUNDEFINEDIOPAIR

In order to satisfy condition ϕ_5^{H} of Lemma 6.3.1, implementations of function HANDLEUNDEFINEDIOPAIR(M, V, q, x, y, TS, G) must ensure that any FSM $I \in \mathcal{F}(M, m)$ that passes the resulting test suite exhibits $\upsilon_q.(x/y)$ in its language if and only if the reference model does. The W, Wp, HSI and H-Methods ensure this by simply adding $\upsilon_q.(x/y)$ to the test suite, while the SPY and SPYH-Methods may alternatively append x/y after some convergent trace $\alpha \in [\upsilon_q]_{TS}$ via function DISTRIBUTEEXTENSION (Algorithm 1). Analogous to configuring whether to exploit convergence in HANDLEUT-STATIC, this latter function may also be employed to implement the behaviour required by the W, Wp, HSI and H-Methods by selecting an EMPTY implementation of convergence graphs. Algorithm 29 thus provides a suitable implementation of HANDLEUNDEFINEDIOPAIR for all considered strategies.

Algorithm 29: handleUndefinedIOPair(M, V, q, x, y, TS, G)

```
1 distributeExtension(υ_q, (x/y), TS, G)
2 return TS
```

6.4 Framework Based on the SPY-Condition

In this section, I develop a variation on the H-Framework, introduced in the previous section, that employs the SPY-Condition (Corollary 4.2.2) in its completeness proofs. While higher order function H-Framework covers all strategies described in Section 5.1, it is based entirely on the (abstract) H-Condition, that is, on establishing preservation of divergence over certain sets of traces. Therefore, it may not be able to easily cover hypothetical new strategies based on the SPY-Condition, which may be developed if new sufficient conditions for preservation of convergence are found.

The new framework, called *SPY-Framework*, is designed to cover the W, Wp, HSI, SPY and SPYH-Methods. It shares with H-Framework the procedural parameters getStateCover, handleStateCover, and sortTransitions. These are to perform the exact same steps and satisfy the same properties as in H-Framework. Additionally, procedural parameter handleIOPair is very similar to handleUndefinedIOPair, with the former being expected to also return a valid convergence graph, as it is no longer used only in the handling of undefined IO-pairs. Parameter handleUndefinedTransition of H-Framework is replaced by parameter establishConvergence, which is expected to update a test suite so that is establishes convergence of a $\upsilon_q.(x/y)$ and $\upsilon_{q'}$ for a given transition (q, x, y, q') of the reference model.

Function SPY-Framework (Algorithm 30) implements this framework. It differs from H-Framework (Algorithm 17) only in the first `foreach`-loop. Where H-Framework employs handleUndefinedTransition to establish preservation of divergence for the extensions of unverified transitions, Algorithm 30 employs establishConvergence in order to iteratively create a transition cover satisfying the SPY-Condition (Corollary 4.2.2).

The following lemma formalises the properties to be satisfied by implementations of the procedural parameters and shows that applying suitable implementations to the procedural parameters results in a strategy for generating complete test suites. This lemma differs from Lemma 6.3.1 only in the fourth and fifth condition.

Lemma 6.4.1. *Let $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ be an FSM with $|Q| = n \leq m$. Suppose that the following invariants hold over the arguments of Algorithm 30:*

ϕ_1^{SPY}) getStateCover(M) *returns a state cover of M.*

ϕ_2^{SPY}) *For all TS, G, V, it holds that if* handleStateCover(M, V) *returns (TS, G), then*

(a) V is $\mathcal{F}(M, m)_{TS}$-divergence-preserving, and

Algorithm 30: SPY-Framework

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Input : function getStateCover
Input : function handleStateCover
Input : function sortTransitions
Input : function establishConvergence
Input : function handleIOPair
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
1  V ← getStateCover(M)
2  (TS, G) ← handleStateCover(M, V)
3  U ← {(q, x, y, q') ∈ h_M | υ_q.(x/y) ≠ υ_q'}  // unverified transitions
4  U ← sortTransitions(U,V)
5  foreach t = (q, x, y, q') ∈ U do
6  |   (TS, G) ← handleIOPair(M, V, q, x, y, TS, G)
7  |   (TS, G) ← establishConvergence(M, V, t, m, TS, G)
8  |   CG-merge(G, υ_q.(x/y), υ_q')
9  foreach q ∈ Q, x ∈ Σ_I, y ∈ Σ_O such that x/y ∉ L_M(q) do
10 |   (TS, G) ← handleIOPair(M, V, q, x, y, TS, G)
11 return TS
```

(b) $\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_V$.

(c) G *is a valid convergence graph for* M *and* $\mathcal{F}(M,m)_{TS}$

ϕ_3^{SPY}) *For all* U, V, sortTransitions(U,V) *returns the same set of transitions as* U.

ϕ_4^{SPY}) *For all* TS, TS', G, G', *transitions* $t = (q,x,y,q') \in h_M$, *and state cover* V *of* M, *if*

- establishConvergence(M, V, t, m, TS, G) *returns* (TS', G'),
- TS *contains the test suite of* handleStateCover(M, V),
- V *is* $\mathcal{F}(M,m)_{TS}$*-divergence-preserving,*
- $\mathcal{F}(M,m)_{TS} \subseteq \mathcal{F}(M,m)_V$, *and*
- G *is a valid convergence graph for* M *and* $\mathcal{F}(M,m)_{TS}$,

then all of the following holds:

(a) $pref(TS) \subseteq pref(TS')$,

(b) $\{\upsilon_q.(x/y), \upsilon_{q'}\}$ *is* $\mathcal{F}(M,m)_{TS'}$*-convergence-preserving, and*

(c) G' *is a valid convergence graph for* M *and* $\mathcal{F}(M,m)_{TS'}$

ϕ_5^{SPY}) *For all* TS, TS', G, G'*, state covers* V *of* M *and* $q \in Q, x \in \Sigma_I, y \in \Sigma_O$*, it holds that if* HANDLEIOPAIR(M, V, q, x, y, TS, G) *returns* (TS', G') *and* G *is a valid convergence graph for* M *and* $\mathcal{F}(M,m)_{TS}$*, then*

- $pref(TS) \subseteq pref(TS')$,
- TS *contains some* $\gamma.(x/y)$ *where* $\gamma \in [v_q]_{TS}$*, and*
- G *is a valid convergence graph for* M *and* $\mathcal{F}(M,m)_{TS}$.

Then application of SPY-FRAMEWORK *on* M*,* m*, and functions* GETSTATECOVER*,* HANDLESTATECOVER*,* SORTTRANSITIONS*,* ESTABLISHCONVERGENCE*, and* HANDLEIOPAIR *results in a test suite* TS *such that* $pref(TS)$ *is complete for testing language-equivalence against reference model* M *over fault domain* $\mathcal{F}(M,m)$. ⊣

Analogous to Lemma 6.3.1, this lemma is proven by establishing that the returned test suite satisfies the corresponding sufficient condition for completeness – in this case the SPY-Condition (Corollary 4.2.2) – as any collection of implementations supplied to the procedural parameters that satisfy ϕ_1^{SPY} to ϕ_5^{SPY} is sufficient to create the transition cover required by that condition.

Interpreting conditions ϕ_1^{SPY} to ϕ_5^{SPY} as predicates over the procedural parameters, Lemma 6.4.1 constitutes the property

$$\begin{aligned}\forall I_1, I_2, I_3, I_4, I_5.\, \phi_1^{\text{SPY}}(I_1) \wedge \phi_2^{\text{SPY}}(I_2) \wedge \phi_3^{\text{SPY}}(I_3) \wedge \phi_4^{\text{SPY}}(I_2, I_4) \wedge \phi_5^{\text{SPY}}(I_5) \\ \longrightarrow \text{SPY-FRAMEWORK}(M, m, I_1, I_2, I_3, I_4, I_5) \\ \text{is complete for } \mathcal{F}(M,m) \text{ w.r.t. } M\end{aligned}$$

Thus, analogously to the H-Framework, proving completeness of implementations using the SPY-Framework reduces to establishing properties of the functions passed to the procedural parameters.

As stated above, Lemma 6.3.1 and Lemma 6.4.1 coincide in all conditions except ϕ_4^{SPY} and ϕ_5^{SPY}. Hence, implementations of procedural parameters GETSTATECOVER, HANDLESTATECOVER, and SORTTRANSITIONS, discussed in Section 6.3, may be reused here. Furthermore, HANDLEIOPAIR may be implemented analogously to HANDLEUNDEFINEDIOPAIR in the H-Framework (see Subsection 6.3.5) via Algorithm 29, additionally returning the possibly modified convergence graph obtained by calling DISTRIBUTEEXTENSION (Algorithm 1). This algorithm satisfies condition ϕ_5^{SPY} of Lemma 6.4.1, as it already satisfies condition ϕ_5^{H} of Lemma 6.3.1, appends x/y to some trace converging with v_q as required, and returns a valid convergence graph by the assumed properties on the preservation of validity of such graphs as detailed in Section 3.3. In the remainder of this section, it therefore remains only to provide implementations for ESTABLISHCONVERGENCE.

6.4.1 ESTABLISHCONVERGENCE

Implementations of ESTABLISHCONVERGENCE required to represent the W, Wp, HSI, SPY, and SPYH-Methods are closely related to the respective implemen-

tations for these strategies of HANDLEUNVERIFIEDTRANSITION (see Subsection 6.3.4). Due to the change in the fourth condition ϕ_4^{H} from Lemma 6.3.1 to ϕ_4^{SPY} of Lemma 6.4.1, implementations of ESTABLISHCONVERGENCE are not required to perform merging of convergent classes themselves. Instead, merging is always performed in line 8 of SPY-FRAMEWORK. This contrasts with H-FRAMEWORK, which considers merging only in some implementations of HANDLEUNVERIFIEDTRANSITION.

As discussed in Subsection 6.3.4, the W, Wp, HSI, and SPY-Methods all establish convergence by the same overall procedure of appending sets of distinguishing traces, differing only in the selection of sets they apply and in whether they distribute over convergent traces. Thus, an implementation of ESTABLISHCONVERGENCE for these strategies is obtained by dropping the merging operation from Algorithm 25, resulting in ESTABLISHCONV-STATIC implemented in Algorithm 31. Function ESTABLISHCONV-STATIC satisfies condition ϕ_4^{SPY} of Lemma 6.4.1 by an argument analogous to that of HANDLEUT-STATIC satisfying condition ϕ_4^{H} of Lemma 6.3.1.

Algorithm 31: ESTABLISHCONV-STATIC

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : transition $(q, x, y, q) \in h_M$
Input : integer m
Input : test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$
Input : convergence graph G
Input : function GETDISTSETFORLENGTH
Output: test suite $TS' \subseteq (\Sigma_I \times \Sigma_O)^*$ and convergence graph G'

```
1  l ← m − n
2  foreach ω ∈ ⋃_{i=0}^{l} (Σ_I × Σ_O)^i such that pref(ω) \ {ω} ⊆ L_M(q') do
3      if ω ∈ L_M(q') then
4          foreach γ ∈ GETDISTSETFORLENGTH(M, q'-after-ω, l, |ω|) do
5              DISTRIBUTEEXTENSION(M, υ_q, (x/y).ω.γ, TS, G)
6              DISTRIBUTEEXTENSION(M, υ_q', ω.γ, TS, G)
7      else
8          DISTRIBUTEEXTENSION(M, υ_q, (x/y).ω, TS, G)
9          DISTRIBUTEEXTENSION(M, υ_q', ω, TS, G)
10 return (TS, G)
```

In the case of the SPYH-Method, an implementation may be derived from HANDLEUT-DYNAMIC (Algorithm 27). Here, it is possible to omit several lines, as it is no longer necessary to support the H or $\mathrm{S}_{partial}$-Methods, which do not establish convergence for each transition. More precisely, the implementation ESTABLISHCONV-DYNAMIC given in Algorithm 32 is derived from Algorithm 27 by dropping the procedural parameter DOESTABLISHCONVERGENCE and the `else`-branch, as well as lines 2 and 5, which are already performed via SPY-

FRAMEWORK. Analogous to the implementation for the W, Wp, HSI, and SPY-Methods, the argument for HANDLEUT-DYNAMIC satisfying condition ϕ_4^{H} of Lemma 6.3.1 also establishes that ESTABLISHCONV-DYNAMIC satisfies condition ϕ_4^{SPY} of Lemma 6.4.1.

Algorithm 32: ESTABLISHCONV-DYNAMIC

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : transition $(q, x, y, q) \in h_M$
Input : integer m
Input : test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$
Input : convergence graph G
Input : function GETDISTSETFORLENGTH
Output: test suite $TS' \subseteq (\Sigma_I \times \Sigma_O)^*$ and convergence graph G'

```
1 S ← V
2 l ← m − n
3 DISTINGUISHFROMSET(υ_q.(x/y), υ_q', V, S, TS, G, l)
4 return (TS, G)
```

Fig. 6.4 summarises how ESTABLISHCONVERGENCE may be implemented for the various strategies, using different implementations of the convergence graph to enable or disable distribution over convergent traces.

6.5 Framework Based on Separable Pairs

In this section, I introduce a framework designed to cover only those strategies that are similar to the H-Method and do not employ distribution over convergent traces. This is motivated by two aspects. First, the implementation of the H-Method via the H-Framework is restricted in the order in which pairs of traces are to be separated. For example, H-FRAMEWORK requires implementations to first separate pairs of traces in the state cover and only thereafter considers extensions of the state cover. The H-Method presented in Algorithm 5 is less restrictive, as it does not prescribe any order in which the pairs in $A \cup B \cup C$ are to be handled. Second, there exist strategies for related conformance relations that closely resemble the H-Method, differing only in the sets of pairs to separate. For example, the H-Method, the *safety-complete* H-Method described in [47], and the exhaustive strategy for requirements-based testing presented in [49] all compute test suites in the following three steps:

1. Compute an initial test suite TS containing a state cover V of reference model M extended by traces of length up to $m - n + 1$.

2. Compute a set of pairs of traces to separate in M. In the H-Method, these are $A \cup B \cup C$ as described in the H-Condition (Lemma 4.1.7). The other strategies employ abstractions of M.

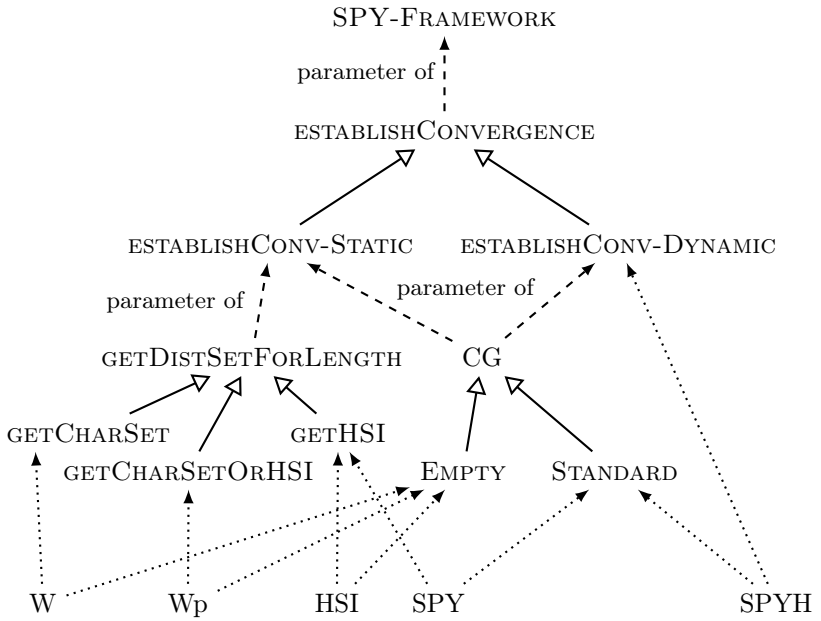


Figure 6.4: Overview of how various strategies may implement parameter establishConvergence of SPY-Framework (Algorithm 30). An arrow from A to B with a white triangle head indicates that B is a possible implementation of A. Dotted arrows indicate which implementations are employed by each strategy. CG abbreviates the choice of implementation for convergence graphs.

3. For each such pair (α, β), find one or more $\gamma \in \Delta_M(M\text{-after-}\alpha, M\text{-after-}\beta)$ and add $\alpha.\gamma$ and $\beta.\gamma$ to the test suite.

These steps constitute the *Pair-Framework* implemented as Pair-Framework in Algorithm 33. By allowing more than a single distinguishing trace to be appended after considered pairs, this framework is also suited to implement the W and HSI-Methods.

Algorithm 33: Pair-Framework

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$
Input : integer m
Input : function getInitialTestSuite
Input : function getPairs
Input : function getSeparatingTraces
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
1 TS ← getInitialTestSuite(M, m)
2 D ← getPairs(M, m)
3 foreach (α, β) ∈ D do
4 |   W ← getSeparatingTraces(M, α, β, TS)
5 |   TS ← TS ∪ {α, β}.W
6 return TS
```

Completeness of test suites for language-equivalence testing implemented via Pair-Framework can be established via the H-Condition as described in the following lemma, which introduces sufficient properties for implementations of the procedural parameters.

Lemma 6.5.1. *Let $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ be an FSM with with $|Q| = n \leq m$. let X_q, A, B, and C be defined as follows*

$$X_q := \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$$
$$A := V \times V$$
$$B := V \times \{v_q.\alpha \mid q \in Q \wedge \alpha \in X_q\}$$
$$C := \{(v_q.\alpha, v.\beta) \mid q \in Q \wedge \beta X_q \wedge \alpha \in \mathit{pref}(\beta) \setminus \{\beta\}\}\}$$

Suppose that there exists a state cover V of M such that following invariants hold over the arguments of Algorithm 33:

ϕ_1^{Pair}) getInitialTestSuite *computes the initial test suite required by the H-Condition:*

$$V \cup \{v_q.\alpha \mid q \in Q \wedge \alpha \in X_q\} \subseteq \text{getInitialTestSuite}(M)$$

ϕ_2^{Pair}) getPairs *computes the pairs required by the H-Condition:*

$$\{(\alpha, \beta) \in A \cup B \cup C \mid \alpha, \beta \in \mathcal{L}(M) \wedge M\text{-after-}\alpha \neq M\text{-after-}\beta\}$$
$$\subseteq \text{getPairs}(M, m)$$

ϕ_3^{Pair}) getSeparatingTraces *contains at least one distinguishing trace for relevant pairs:*

$$\begin{aligned}&\forall TS, \alpha, \beta.\\ &\quad (\alpha, \beta \in \mathcal{L}(M) \wedge \alpha, \beta \textit{ are not } TS\textit{-separable})\\ &\quad \longrightarrow \Delta_M(M\textit{-after-}\alpha, M\textit{-after-}\beta)\\ &\qquad \cap \text{getSeparatingTraces}(M, \alpha, \beta, TS) \neq \emptyset\end{aligned}$$

Then, the application of Pair-Framework *to values M, m, and functions* getInitialTestSuite, getPairs, *and* getSeparatingTraces *results in a test suite complete for language-equivalence testing of M against $\mathcal{F}(M, m)$.* ⊣

Proof. This result follows from Algorithm 33 generating a test suite satisfying the weakened H-Condition (Lemma 4.1.3) if supplied with implementations of its procedural parameters satisfying ϕ_1^{Pair} to ϕ_3^{Pair}. □

6.5.1 getInitialTestSuite and getPairs

Simple implementations for procedural parameters getInitialTestSuite and getPairs may be obtained by explicitly generating the smallest sets required to satisfy conditions ϕ_1^{Pair} and ϕ_2^{Pair} of Lemma 6.5.1, respectively. This requires both implementations to share a state cover for the reference model, which may be supplied directly or via a suitable procedural parameter for a function computing a state cover. Algorithms 34 and 35 provide implementations following the latter approach. Thereby, and by independently computing the X_q sets, these two implementations are not fully compliant to the objective of minimising code duplication as described in Section 6.2. In the case of the Pair-Framework, this is justified, as it is designed to also accommodate strategies using sets of pairs of traces to separate selected by criteria other than those of the H-Condition. Furthermore, code refinement as discussed in Section 12.1 allows avoiding this duplication in the Isabelle/HOL implementations of these functions. Function getStateCoverByBFS (Algorithm 18) discussed in Subsection 6.3.1 constitutes a suitable implementation for procedural parameter getStateCover.

Algorithm 34: getInitialTestSuite-H

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Input : function getStateCover
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

$V \leftarrow \text{getStateCover}(M)$
foreach $q \in Q$ **do**
  $X_q \leftarrow \{v_q.\alpha.(x/y) \mid \alpha \in \mathcal{L}_M(q) \wedge |\alpha| \leq m - n \wedge x \in \Sigma_I \wedge y \in \Sigma_O\}$
return $V \cup \{v_q.\alpha \mid q \in Q \wedge \alpha \in X_q\}$

Algorithm 35: getPairs-H

Input : minimal OFSM $M = (Q, q_0, \Sigma_I, \Sigma_I, h)$ with $|Q| = n$
Input : integer m
Input : function getStateCover
Output: test suite $TS \subseteq (\Sigma_I \times \Sigma_O)^*$

```
1 V ← getStateCover(M)
2 foreach q ∈ Q do
3     X_q ← {υ_q.α.(x/y) | α ∈ L_M(q) ∧ |α| ≤ m − n ∧ x ∈ Σ_I ∧ y ∈ Σ_O}
4 A ← V × V
5 B ← V × {υ_q.α | q ∈ Q ∧ α ∈ X_q}
6 C ← {(υ_q.α, υ.β) | q ∈ Q ∧ βX_q ∧ α ∈ pref(β)}
7 return {(α, β) ∈ A ∪ B ∪ C | α, β ∈ L(M) ∧ M-after-α ≠ M-after-β}
```

6.5.2 getSeparatingTraces

Whereas the W, HSI, and H-Methods share the same selection of initial test suite and pairs to separate, they differ in the sets of traces applied in order to separate. First, the W-Method simply applies the characterisation set after every trace to be separated from another trace. Thus, the W-Method supplies to procedural parameter getSeparatingTraces a function such as getCharSet' implemented in Algorithm 36. This function performs the same computation as getCharSet (Algorithm 23) discussed in Subsection 6.3.2, differing from the latter only in its parameters, of which all but reference model M are ignored.

Algorithm 36: getCharSet'(M, α, β, TS)

```
1 return {getShortestDistTrace(M, q, q') | q, q' ∈ Q ∧ q ≠ q'}
```

Implementations of the HSI-Method must instead supply a function that effectively applies fixed harmonised state covers. Note here that function PairFramework appends the result of a call getSeparatingTraces(M, α, β, TS) to both α and β. Thus, for the HSI-Method it is not sufficient to implement getSeparatingTraces by returning the harmonised state identifier of M-after-α or M-after-β, as these may not be identical. Note furthermore, that by using the pairs generated via getPairs-H, each α or β on which getSeparatingTraces(M, α, β, TS) is called must be separated from the traces in the state cover. Therefore, application of a harmonised state identifier $H_{M\text{-after-}\alpha}$ after some α may be distributed over all calls to getSeparatingTraces where α is passed as second or third parameter. This results in a harmonised state identifier analogous to the result of function getHSI (Algorithm 22). Using this approach, function getDistTrace, implemented in Algorithm 37, constitutes an implementation of getSeparatingTraces for the HSI-Method.

Finally, the H-Method may implement getSeparatingTraces with some function that returns an empty set if the provided test suite is already sufficient

Algorithm 37: getDistTrace(M, α, β, TS)

```
1 γ ← getShortestDistTrace(M, M-after-α, M-after-β)
2 return {γ}
```

to separate the traces. This is implemented in getDistTraceIfReq (Algorithm 38), which falls back to getDistTrace if the test suite is not sufficient and a distinguishing trace needs to be added. Function getDistTraceIfReq is a very simple implementation that does not perform any heuristics in finding a distinguishing trace that requires the least additions to the test suite, for example by trying to find some distinguishing trace already appended to one of the traces to separate. More sophisticated implementations might adapt function bestPrefixOfSepSeq (Algorithm 13) for this purpose. Chapter 14 compares the performance of two implementations of the H-Method, respectively employing getDistTraceIfReq and a more elaborate heuristic.

Algorithm 38: getDistTraceIfReq(M, α, β, TS)

```
1 if α and β are TS-separable then
2 |   return ∅
3 else
4 |   return getDistTrace(M, α, β, TS)
```

Figure 6.5 summarises how the W, HSI, and H-Methods may be implemented via function Pair-Framework.

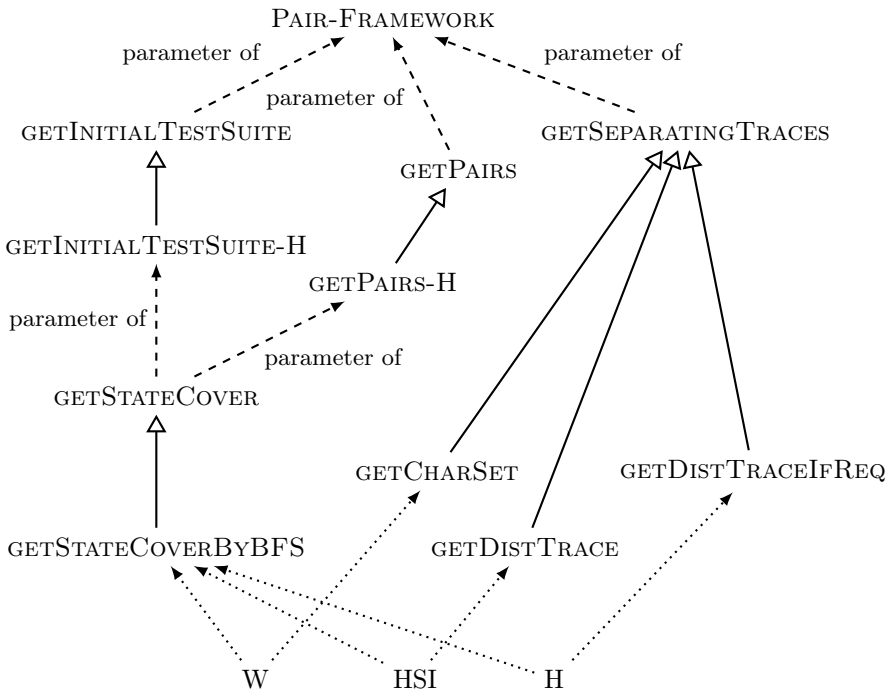


Figure 6.5: Overview of how various strategies may implement the procedural parameter of Pair-Framework (Algorithm 33). An arrow from A to B with a white triangle head indicates that B is a possible implementation of A. Dotted arrows indicate which implementations are employed by each strategy. As all considered strategies employ functions getInitialTestSuite-H and getPairs-H, the corresponding dotted arrows have been omitted.

Part III

Formalisation in Isabelle/HOL

Chapter 7

Overview

In this part, I describe mechanised formalisations of the frameworks developed in Chapter 6, implemented using the generic proof assistant Isabelle[1]. These formalisations are stored in several so-called theory files (with file extension `thy`), which in turn contain definitions of functions and data types, lemmata, and instructions to the code generator (see Section 12.1). The theory files developed for this work, as well as instructions on how to use them via Isabelle are publicly available in the repository accompanying this work (see Section 1.4). The subsequent descriptions of these theory files contain various keywords and concepts of Isabelle and in particular of Isabelle/HOL, a higher-order logic theorem proving environment for Isabelle. I describe relevant keywords at their first occurrence in code listings. For a comprehensive overview see the reference manual [119], which is included in current Isabelle distributions, as are several tutorials, including [6, 39, 40, 61, 76]. See [59] for another perspective on first steps with Isabelle/HOL. Some best practices for the design and maintenance of mechanised proofs are developed in [16, 123]. Further aspects of *proof engineering* are discussed in [3, 56, 94, 109].

Figure 7.1 provides an overview over the theory files that describe and prove complete implementations of the W, Wp, HSI, H, SPY, SPYH and $S_{partial}$-Methods using the H, SPY and Pair-Frameworks[2]. The theory files may be grouped as follows:

First, `Util.thy` and `Prefix_Tree.thy` provide results not immediately related to finite state machines. More precisely, `Util.thy` collects a large number of auxiliary lemmata on functions already defined within Isabelle/HOL, as well as some basic functions on lists, while `Prefix_Tree.thy` introduces a tree data structure suitable for representing finite prefix-closed sets of IO-traces. While extensive, these two theory files do mostly contain very basic results. For this

[1] https://isabelle.in.tum.de

[2] A complete dependency graph including theory files employed for refinements as well as imported theory files of the *Archive of Formal Proofs* (https://www.isa-afp.org) and files imported from the standard library of Isabelle/HOL is part of the publicly available repository for the present work (see Section 1.4).

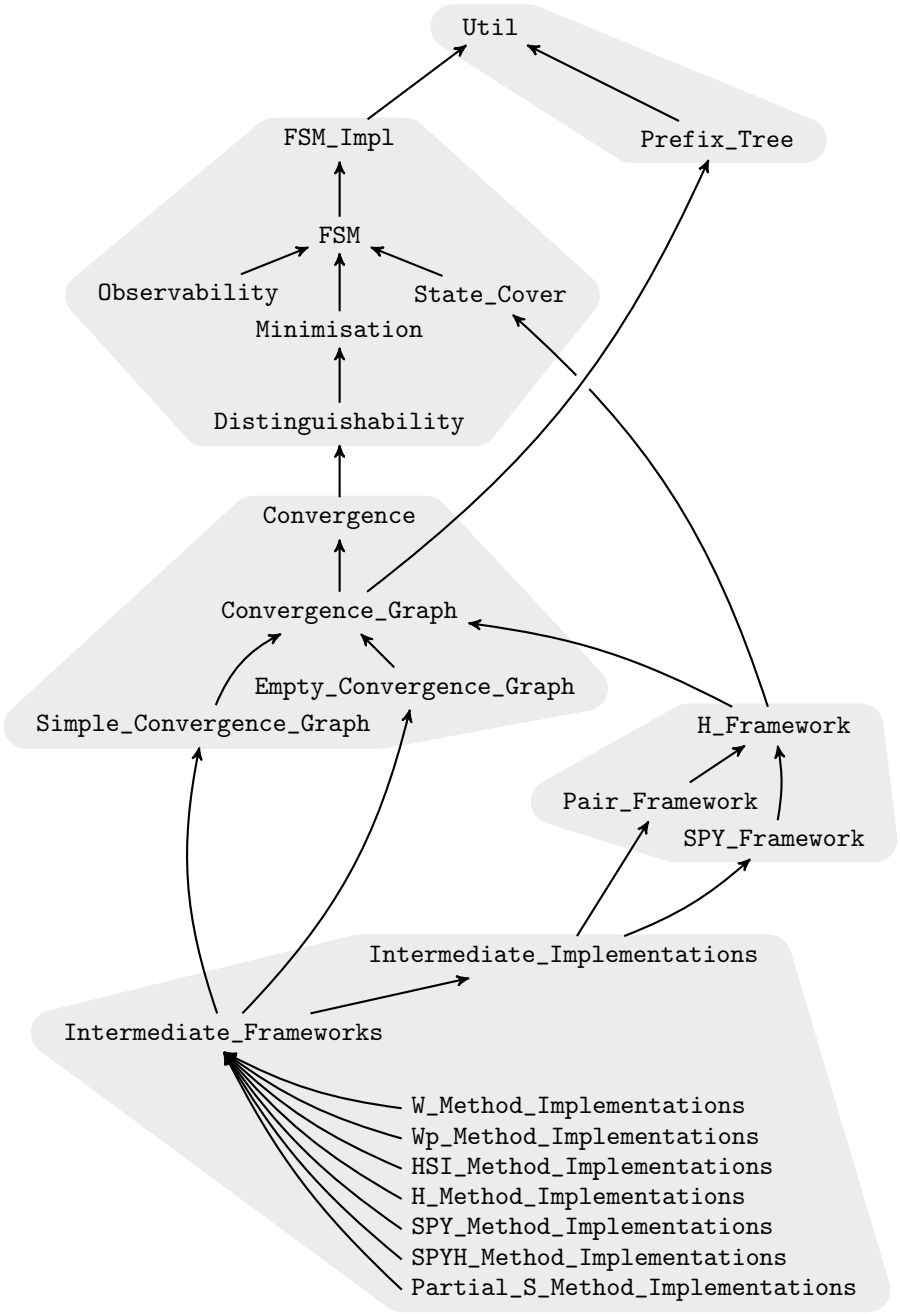


Figure 7.1: Simplified dependency graph of the main theory files constituting the mechanisation of the frameworks described in Chapter 6. The file extension `thy` is omitted.

reason, in the following I reference `Util.thy` only in the context of more elaborate results in other theory files that employ it, while an overview of the data structure `prefix_tree` and its operations as developed in `Prefix_Tree.thy` is provided in Appendix C.

Second, finite state machines as described in Chapter 2 are introduced in theory files `FSM_Impl.thy` and `FSM.thy`, with further basic operations and notions being mechanised in files `Observability.thy` (describing a language-preserving transformation to establish observability), `State_Cover.thy` (describing state covers and simple breadth-first-search computations of them), `Minimisation.thy` (describing a language-preserving minimisation algorithm) and `Distinguishability.thy` (formalising an algorithm to compute minimal length distinguishing traces). This group of files is described in further detail in Chapter 8.

The next group contains all files describing convergence and convergence graphs. This group consists of files `Convergence.thy` (defining the notion of convergence), `Convergence_Graph.thy` (describing the operations required on implementations of convergence graphs and formalising the notion of validity as described in Section 3.3), and `Empty_Convergence_Graph.thy` as well as `Simple_Convergence_Graph.thy`, implementing convergence graphs Empty and Standard as employed in Chapter 6. Chapter 9 describes these files in further detail.

These first three groups provide all definitions and lemmata required to define and then prove complete the frameworks developed in Chapter 6. Each of the three frameworks is handled in a single theory file of the same name and is discussed in Chapter 10.

Finally, implementations of the procedural parameters of the frameworks are contained in theory file `Intermediate_Implementations.thy`, from which I derive in `Intermediate_Frameworks.thy` partial applications of the frameworks. For example, implementations of the W, Wp, and HSI-Methods via SPY-Framework all use establishConvergence-Static (Algorithm 31) and the Empty convergence graph, differing solely in the implementations supplied to the single procedural parameter getDistSetForLength of function establishConvergence-Static. Thus, by providing a function that applies establishConvergence-Static and Empty to SPY-Framework and which has as procedural parameter only getDistSetForLength, it is possible to reduce repetition of code in the implementation of these three test strategies. I provide a separate theory file for each considered test strategy, containing all implementations of the respective strategy via the applicable frameworks. Together with the intermediate implementations and frameworks, these are presented in Chapter 11.

7.1 Proof Effort

Table 7.1 provides an overview of the sizes of the developed theory files in terms of lines of code, number of definitions and number of proven lemmata. Note

that this table does not include several theory files that solely contribute to the generation of executable implementations and their optimisation, or theory files describing strategies for the reduction conformance relation. These are described in further detail in Part IV and Appendix A, respectively. Appendix D provides an overview of relevant functions defined in the theory files listed in Table 7.1.

In formalising and implementing the frameworks discussed in Chapter 6, Isabelle/HOL has proven to be capable of proving a large number of lemmata correct automatically. This is shown with a green bar in Fig. 7.2 and column (b_s) of Table 7.2, listing the number of lemmata which are each proven entirely by a single invocation of automatic proof methods. Consider for example the following lemma of name `filter_map_elem` from theory file `Util.thy`:

```
lemma filter_map_elem : "y ∈ set (map g (filter f xs)) ⟹
                         ∃ x ∈ set xs . f x ∧ y = g x"
  by auto
```

This lemma formalises the fact that if some element y is contained in the list obtained from some list xs by first reducing xs to those elements satisfying predicate f and then mapping function g over the result, then there must exist an element x in the original list xs such that x satisfies f and $g(x)$ is equivalent to y[3,4,5]. The lemma is proven via `by auto`, which instructs Isabelle to try to prove the lemma by a single application of proof method `auto`. No manual guidance of the proof or elaboration of intermediate proof steps is required. See Section 9.4.4 of [119] for an overview of fully automated proof methods available in Isabelle.

In this category of proofs I also include lemmata for which *Sledgehammer* has found a single line proof. Sledgehammer is a tool provided with Isabelle distributions that applies various automated theorem provers and satisfiability-modulo-theories solvers (SMT-solvers) in parallel to translations of proof goals in Isabelle/HOL. If any of the applied provers or solvers finds a solution (that is, a proof), then it is attempted to reconstruct this proof in Isabelle (see [79]), for example by identifying the facts employed in the solution and supplying these to the automatic theorem prover Metis (see [53]) that is part of Isabelle and whose proofs are verified by the Isabelle kernel. See [6, 7, 8] for further details on Sledgehammer. Sledgehammer has been evaluated in [11, 77], where its suitability to work with structured proofs has been highlighted[6].

[3] Note that in Isabelle/HOL, function application of $g(x)$ is written without parentheses as `g x`. This extends to functions with more than a single argument, such that `map h zs` applies h to all elements in zs.

[4] In the following, I write Isabelle/HOL expressions in a typewriter font. This also applies to arguments and may result in the use of different fonts for the same identifier. For example, application of some Isabelle/HOL function `f` to some previously introduced x is written as `(f x)`, where x and `x` denote the same object.

[5] Function `set` obtains the set of all elements in a list.

[6] That is, in cases where Sledgehammer and other automated tools fail to find a proof for a given lemma, it has been identified as a suitable workflow for the user to identify intermediate properties sufficient to prove the lemma and then to prove these via Sledgehammer. I share this view, as Sledgehammer has proven able to automatically prove many such intermediate properties in my formalisation, allowing me to focus on higher level proof steps.

Table 7.1: Overview of the sizes of the theory files constituting the mechanisation of the frameworks described in Chapter 6, counting the total lines of code (kLOC), the number of function or type definitions (Def.) and the number of lemmata (Lem.) proven in each file.

Theory File	kLOC	Def.	Lem.
`Convergence.thy`	3.0	4	29
`Convergence_Graph.thy`	0.1	4	2
`Distinguishability.thy`	0.7	7	11
`Empty_Convergence_Graph.thy`	0.1	5	3
`FSM.thy`	6.4	96	300
`FSM_Impl.thy`	0.3	27	5
`H_Framework.thy`	1.8	5	7
`H_Method_Implementations.thy`	1.0	10	18
`HSI_Method_Implementations.thy`	0.2	6	9
`Intermediate_Frameworks.thy`	0.5	10	11
`Intermediate_Implementations.thy`	7.0	35	60
`Minimisation.thy`	1.8	4	42
`Observability.thy`	1.3	3	9
`Pair_Framework.thy`	1.4	6	9
`Partial_S_Method_Implementations.thy`	0.1	4	2
`Prefix_Tree.thy`	2.0	21	57
`Simple_Convergence_Graph.thy`	2.0	20	47
`SPY_Framework.thy`	1.5	3	2
`SPY_Method_Implementations.thy`	0.2	4	6
`SPYH_Method_Implementations.thy`	0.1	4	5
`State_Cover.thy`	0.7	6	14
`Util.thy`	3.8	31	181
`W_Method_Implementations.thy`	0.3	8	12
`Wp_Method_Implementations.thy`	0.3	6	11
Σ	36.6	329	852

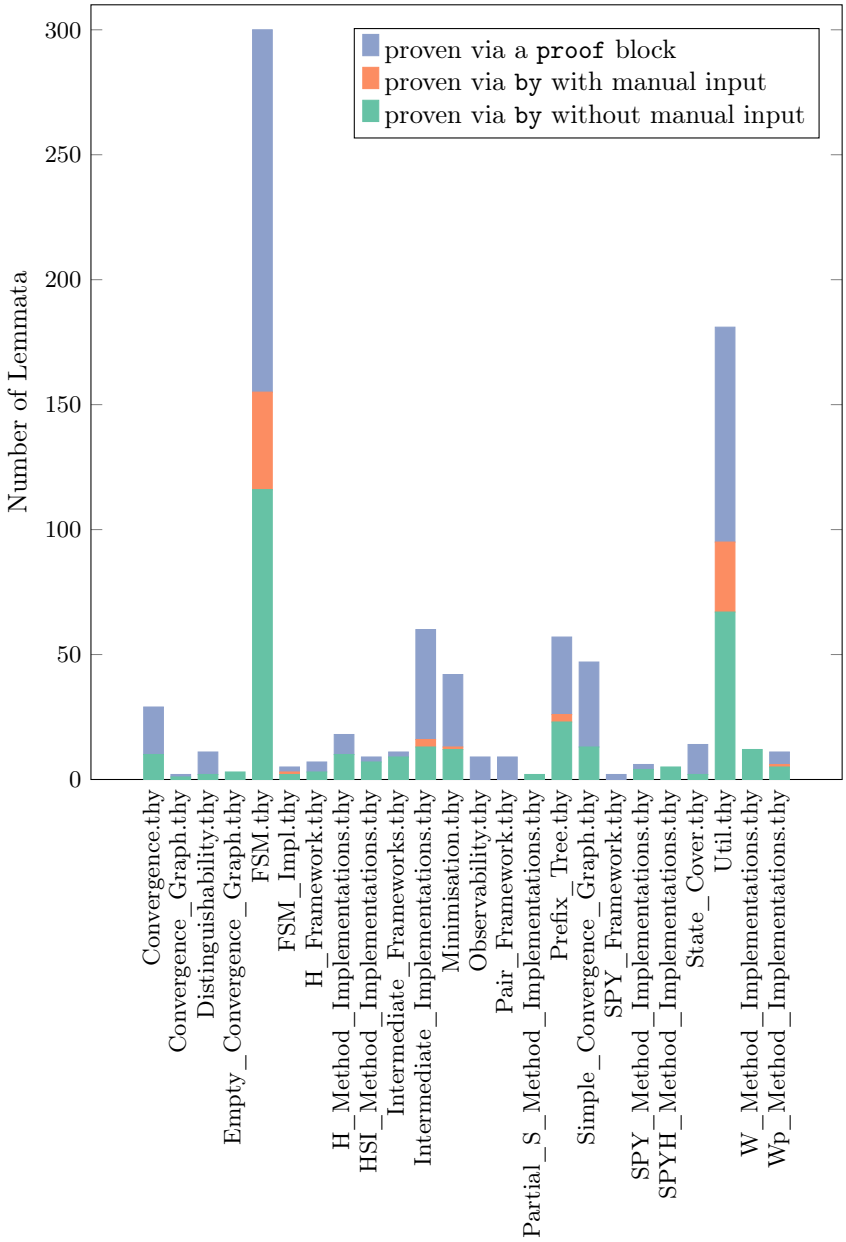


Figure 7.2: Number of lemmata in the theory files discussed in Part III that are proven (a) by a one-line `by` command without further manual input, that is, essentially fully automatically by Isabelle/HOL, (b) by a one-line `by` command augmented with user specified facts or unfolding of some definitions, or (c) by a more complex multi-line `proof` block. The exact numbers of these kinds of lemmata are given in Table 7.2. File extensions have been omitted.

Table 7.2: Overview of the composition of proofs in the theory files discussed in Part III. The following columns are presented: (l) total number of lemmata; (b) lemmata proven via `by` without a proof block; (b_s, b_m, b_c) subsets of b representing proofs without further manual input, the subset thereof using proof method `metis`, and proofs with user supplied facts and unfoldings; (p) lemmata with a `proof` block; (p_i, p_c) number of such proofs using induction and contradiction, respectively; (p_n) proofs with nested `proof` blocks; (d) maximum nesting depth of such proofs. Columns (b_s), (b_c), and (p) respectively correspond to the green, orange, and blue bars in Fig. 7.2.

Theory File	l	b	b_s	b_m	b_c	p	p_i	p_c	p_n	d
`Convergence`	29	10	10	3	0	19	0	0	9	7
`Convergence_Gr.`	2	1	1	0	0	1	0	0	0	-
`Distinguishabi.`	11	2	2	0	0	9	1	0	5	3
`Empty_Converge.`	3	3	3	0	0	0	0	0	0	-
`FSM`	300	155	116	26	39	145	20	3	61	5
`FSM_Impl`	5	3	2	0	1	2	0	0	1	1
`H_Framework`	7	3	3	0	0	4	0	1	4	4
`H_Method_Imple.`	18	10	10	0	0	8	1	0	1	7
`HSI_Method_Imp.`	9	7	7	0	0	2	0	0	2	2
`Intermediate_F.`	11	9	9	0	0	2	0	0	0	-
`Intermediate_I.`	60	16	13	5	3	44	7	0	34	10
`Minimisation`	42	13	12	0	1	29	6	1	20	5
`Observability`	9	0	0	0	0	9	6	0	9	5
`Pair_Framework`	9	0	0	0	0	9	0	0	7	4
`Partial_S_Meth.`	2	2	2	0	0	0	0	0	0	-
`Prefix_Tree`	57	26	23	2	3	31	8	0	20	6
`Simple_Converg.`	47	13	13	0	0	34	11	0	25	5
`SPY_Framework`	2	0	0	0	0	2	0	0	2	6
`SPY_Method_Imp.`	6	4	4	0	0	2	0	0	2	2
`SPYH_Method_Im.`	5	5	5	0	0	0	0	0	0	-
`State_Cover`	14	2	2	1	0	12	0	1	9	5
`Util`	181	95	67	19	28	86	39	2	60	6
`W_Method_Imple.`	12	12	12	1	0	0	0	0	0	-
`Wp_Method_Impl.`	11	6	5	1	1	5	0	0	3	4
Σ	852	397	321	58	76	455	99	8	274	

The following lemma of `Util.thy` constitutes an example of this process, as the facts supplied to the single invocation of `metis` have all been automatically identified by Sledgehammer. The lemma formalises the simple fact that the number of elements obtained by filtering in a prefix of list xs is at most that obtained by filtering xs itself[7].

```
lemma filter_take_length :
  "length (filter P (take i xs)) ≤ length (filter P xs)"
  by (metis append_take_drop_id filter_append le0
        le_add_same_cancel1 length_append)
```

As almost all invocations of `metis` in my formalisation have been derived from applying Sledgehammer, Table 7.2 separately lists the number of proofs consisting only of a single invocation of `metis` in column (b_m).

Next, the orange bar in Fig. 7.2 and column (b_c) in Table 7.2, respectively show the number of lemmata that require only limited manual input, such as providing a rough proof idea (e.g. induction or case analysis), selecting certain facts from the previously established lemmata to use in the proof, or unfolding certain definitions employed in the lemma. Consider the following lemma of `Util.thy`, which formalises the fact that the list obtained by replicating a list xs n times and concatenating the result has length $n \cdot |xs|$:

```
lemma concat_replicate_length :
  "length (concat (replicate n xs)) = n * (length xs)"
  by (induction n; simp)
```

This lemma is easily proven to hold by induction over the number n of replications, since zero replications produce the empty list, which is trivially of length 0, and $(n+1)$ replications are identical to appending xs to n replications of it, which by the induction hypothesis is of length $n \cdot |xs|$, resulting in the desired combined length $(n+1) \cdot |xs|$. Via `by (induction n; simp)`, I merely provide to Isabelle/HOL the overall proof idea, namely the induction on n, while the fully automated proof method `simp` is sufficient to dispatch any arising proof goal. Note here that Sledgehammer would also have been able to find a single-line `metis` proof for this lemma. This does not hold, however, for all lemmata proven in this way (assuming some practical time bound on the invocation of Sledgehammer). Further examples of lemmata in this category are given in Section 8.2.2.

All lemmata of my formalisation not proven via a single invocation of automatic proof methods following keyword `by` are proven via a `proof` block, implementing proofs in the *Isar* (*Intelligible Semi-Automated Reasoning*) proof language introduced in [121]. Note that there exist other styles of proofs, such as chains of `apply` steps (see [119]). I have not employed these in my formalisation.

As a first example for Isar, consider the following lemma of `Util.thy`, which formalises that the cardinality of a set A containing only singletons is equivalent to the cardinality of the union of all sets it contains:

[7] Expression `(take i xs)` returns the first i elements of xs. Note that lists in Isabelle/HOL are finite by construction.

```
lemma card_union_of_singletons :
  assumes "⋀ S . S ∈ A ⟹ (∃ x . S = {x})"
shows "card (⋃ A) = card A"
proof -
  let ?f = "λ x . {x}"
  have "bij_betw ?f (⋃ A) A"
    unfolding bij_betw_def inj_on_def using assms by fastforce
  then show ?thesis
    using bij_betw_same_card by blast
qed
```

Here, the premise on the structure of A is encoded using the universal quantifier ⋀ as *"for all set S, if S is contained in A, then there exists a value x such that S is the singleton $\{x\}$"*, and cardinality is obtained by predefined function `card`. The proof block is bounded by the initiating keyword `proof` and the closing `qed`. It establishes the lemma by defining a function `?f` that wraps its inputs into a singleton, followed by showing that this function constitutes a bijection between set A and the union of all sets in A, encoded using predefined predicate `bij_betw`. This is sufficient to prove the cardinalities equal by using the predefined lemma `bij_betw_same_card`.

The number of lemmata whose proofs are `proof` blocks are indicated as the blue bars in Fig. 7.2 and column (c) in Table 7.2. Note that intermediate results in a proof block may themselves be established via a proof block, allowing the nesting of proofs. Proof methods employed in such intermediate proofs are not considered in Fig. 7.2 and Table 7.2. Table 7.2 lists the number of lemmata whose proof blocks contain nested proof blocks in column (p_n) and gives the maximum depth of such nesting in column (d). The maximum nesting depth of 10 occurs in a correctness proof for bestPrefixOfSepSeq (Algorithm 13), resulting from several nested case distinctions. Column (p_n) includes proofs ranging from the 5 lines of `card_union_of_singletons` to several hundred lines in the proofs of non-trivial results such as the H-Condition (Definition 4.1.3, formalised in Section 10.1).

Chapter 8

Finite State Machines

In this chapter, I introduce the data types employed to represent finite state machines, as well as the mechanised formalisations of several basic operations and properties of FSMs. To this end, Section 8.1 describes the unconstrained data type `fsm_impl` which is augmented to type `fsm` via a well-formedness constraint in Section 8.2. The latter section also describes formalisations of the basic properties of FSMs introduced in Chapter 2, such as paths, language, and observability. Thereafter, Section 8.3 formalises state covers and presents a simple breadth-first-search based algorithm of computing state covers. Next, in Section 8.4 a classical algorithm for transforming an FSM into a language-equivalent observable FSM is implemented and proven correct. Section 8.5 formalises the concept of *OFSM-Tables* as introduced in [82], generalising so-called P_k-Tables (see [37]) to observable FSMs. OFSM-Tables are then employed to transform an observable FSM into a language-equivalent minimal observable FSM by partitioning the states into classes of language-equivalent states. Finally, Section 8.6 formalises the use of these tables to compute distinguishing traces of minimal length for a given pair of distinct states.

An overview of relevant functions defined in theory files discussed in this chapter is given in Appendices D.1 to D.6.

8.1 Underlying Data Type for FSMs

As underlying representation of finite state machines, I have chosen a simple data type `fsm_impl` defined as follows in `FSM_Impl.thy`:

```
datatype ('state, 'input, 'output) fsm_impl
    = FSMI (initial     : 'state)
           (states      : "'state set")
           (inputs      : "'input set")
           (outputs     : "'output set")
           (transitions : "('state × 'input × 'output × 'state) set")
```

Here the first line via keyword `datatype` introduces a new type constructor

`fsm_impl` that has three type parameters: `'state`, `'input` and `'output`. As their names imply, these type parameters respectively describe the types of states, inputs, and outputs. That is, an FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ with $Q \subset \mathbb{N}$, $\Sigma_I \subset \mathbb{Z}$, and Boolean outputs is of type `(nat,int,bool) fsm`, where `nat`, `int`, and `bool` are the predefined Isabelle/HOL types for natural numbers, integers, and Booleans, respectively. Note that in Isabelle/HOL arguments for the type parameters are provided in front of the type constructor.

After the equals sign, a single constructor `FSMI` is defined with 5 arguments, corresponding to the representation of a finite state machine as a 5-tuple. The 5 components of an FSM may be retrieved via functions `initial`, `states`, `inputs`, `outputs`, `transitions`, respectively obtaining the initial state, state set, input alphabet, output alphabet and transition set of a given FSM. In this definition, `'a set` is the Isabelle/HOL type for (possibly infinite) sets of elements of the same type `'a`, while `'a × 'b` is the type for pairs whose left and right elements are of type `'a` and `'b`, respectively. Tuples of more than 2 elements, such as the type of transitions in the set returned by function `transitions`, are represented as pairs nested to the right. For example, if m is a `(nat,int,bool) fsm`, then `outputs m` returns a `bool set` representing the output alphabet of m, while `transitions m` returns a `(nat × int × bool × nat) set`, which is identical to type `(nat × (int × (bool × nat))) set`.

In Isabelle/HOL, the set $\{a, b, c\}$ may simply be written as `{a,b,c}`, assuming that a, b, c are of the same type. Similarly, a tuple (a, b, c, d) may be written as `(a,b,c,d)`, which is syntactic sugar for `(a,(b,(c,d)))`. Thus, FSM $M_{8.1}$ depicted in Fig. 8.1 may be represented as

```
FSMI 0 {0,1,2,3} {0,1} {0,1} { (0,0,0,1), (0,1,0,0), (0,2,1,2),
                               (1,0,0,1), (1,1,0,1), (1,2,1,0),
                               (2,0,0,1), (2,1,1,3), (2,2,1,2),
                               (3,0,0,1), (3,1,0,3), (3,2,1,1) }
```

Note that, while natural, the above definition of finite state machines is only one of several possible encodings. For example, in [19], Brucker and Wolff represent the transition relation of an automaton over state type `'state`, input type `'input`, and output type `'output` not by type

$$(\texttt{'state} \times \texttt{'input} \times \texttt{'output} \times \texttt{'state})\ \texttt{set}$$

but by the isomorphic type

$$\texttt{'state} \Rightarrow (\texttt{'input} \Rightarrow (\texttt{'output} \times \texttt{'state})\ \texttt{set})$$

I have decided upon the representation as a set of tuples, as it allows for more natural traversal of all transitions of an FSM[1] and lends itself more directly to

[1] Note that Brucker and Wolff in [19] consider automata whose state spaces and hence transition sets are not necessarily finite, in which case the explicit storage as a set is not feasible in practice for standard implementations of sets. In the present work this is not required, as FSMs contain only finitely many transitions (as discussed in the next section).

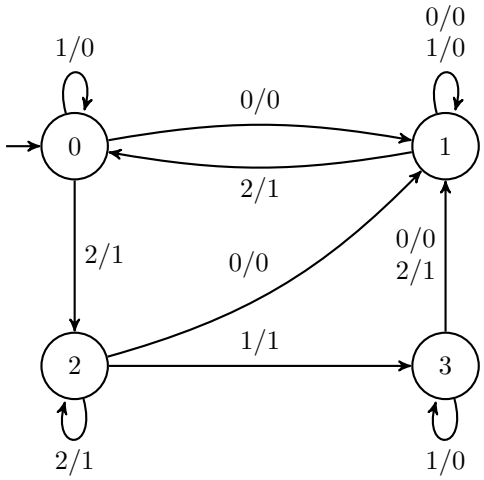


Figure 8.1: FSM $M_{8.1}$.

refinement (see Section 12.4). My formalisation supports viewing the transition relation as a function via functions `h` and `h_obs` introduced later in this section.

To simplify signatures, type synonyms are introduced for transitions (4-tuples consisting of source state, input, output, and target state) and paths (lists of transitions). As exemplified in these type synonyms, I will often employ type variables `'a`, `'b`, `'c` instead of the more expressive but longer `'state`, `'input`, and `'output`.

```
type_synonym ('a,'b,'c) transition = "('a × 'b × 'c × 'a)"

type_synonym ('a,'b,'c) path = "('a,'b,'c) transition list"
```

Thereafter, theory file `FSM_Impl.thy` introduces all low-level functions that directly create or modify values of type `fsm_impl`, such as the creation of FSMs from a given list of transitions, implemented in the following function:

```
fun fsm_impl_from_list :: "'a ⇒
                           ('a,'b,'c) transition list ⇒
                           ('a,'b,'c) fsm_impl"
where
  "fsm_impl_from_list q [] = FSMI q {q} {} {} {}" |
  "fsm_impl_from_list q (t#ts) =
    (let ts' = set (t#ts)
     in FSMI (t_source t)
             ((image t_source ts') ∪ (image t_target ts'))
             (image t_input ts')
             (image t_output ts')
             (ts'))"
```

Here, `fun` is one of several keywords in Isabelle/HOL that serve to define new functions[2]. It is followed by the name of the function to define, here `fsm_impl_from_list`, a double colon, the type signature of the function, keyword `where` and finally a `|`-separated list of defining equations. The type signature specifies the type of the function to define, which here is a function that takes as arguments an `'a` and a `('a,'b,'c) transition list` and returns an `('a,'b,'c) fsm_impl`[3]. Finally, the defining equations may use pattern matching to describe the function to define. For `fsm_impl_from_list`, I separately consider two patterns on the provided list of transitions. In the first defining equation, pattern `[]` is used to match the empty list. In this case, the function uses its first argument, q, to create an `fsm_impl` whose initial and only state is q, and that uses empty sets for input and output alphabets, as well as transitions. The second defining equation considers the case where the provided list of transitions consists of a transition t and a remaining list of transitions ts[4]. In this case, the first argument q is not considered further and an `fsm_impl` is created that uses as initial state the source of transition t, as states, inputs, or outputs respectively the sets of all sources or targets, inputs, or outputs of the provided list of transitions, and finally as transition set a set whose elements are the provided transitions[5].

An example of a function modifying an existing FSM is `add_transition`, which adds a transition to a given FSM if that transition uses only states, inputs, and outputs already defined in the FSM:

```
fun add_transition :: "('a,'b,'c) fsm_impl ⇒
                       ('a,'b,'c) transition ⇒
                       ('a,'b,'c) fsm_impl"
  where
  "add_transition M t =
    (if t_source t ∈ states M ∧ t_input t ∈ inputs M ∧
        t_output t ∈ outputs M ∧ t_target t ∈ states M
     then FSMI (initial M)
               (states M)
               (inputs M)
               (outputs M)
               (insert t (transitions M))
     else M)"
```

[2] More precisely, it is a shorthand notation for a particular usage of keyword `function` as described in [61]. Functions in Isabelle/HOL are required to be total, which needs to be proven at the end of a function definition performed via `function`. Keyword `fun` provides default methods to attempt to automatically prove termination (and also completeness and compatibility of patterns, see [61]). These have proven sufficient for most functions in the present work. A partial example for a manual termination proof is given in Subsection 8.5.2.

[3] In Isabelle/HOL, `'a ⇒ 'b` is the type of functions that take a single argument of type `'a` and return a value of type `'b`. Type constructor ⇒ is right-associative, such that `('a ⇒ 'b ⇒ 'c)` is identical to `('a ⇒ ('b ⇒ 'c))`

[4] `[]` and # are the constructors of data type `list`. `[]` is the empty list, whereas # takes an `'a` and an `'a list` and prepends the single element to the front of that list.

[5] The predefined function `image` obtains the image of a given function over a given set. That is, expression `(image f A)` evaluates to $\{f(x) \mid x \in A\}$.

Finally, `FSM_Impl.thy` provides functions for interpreting the set of transitions of an FSM as functions mapping states and inputs or IO-pairs to responses and/or reached states. First, function `h` applied to an FSM maps each pair of state and input to all pairs of output and state such that the 4-tuple of these values constitutes a transition:

```
fun h :: "('a,'b,'c) fsm_impl ⇒ ('a × 'b) ⇒ ('c × 'a) set" where
  "h M (q,x) = { (y,q') . (q,x,y,q') ∈ transitions M }"
```

This function uses Isabelle/HOL set-builder notation to build the set of all pairs (y, q') such that $(q, x, y, q') \in$ `transitions M` holds.

Function `h_obs` uses return type `'a option` to encode for given FSM M, state q, input x and output y, whether there exists exactly one state q' such that (q, x, y, q') is a transition of M. If this is the case, then `(Some q')` is returned. Otherwise, `Nothing` is returned.

```
fun h_obs :: "('a,'b,'c) fsm_impl ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'a option" where
  "h_obs M q x y = (let
      tgts = snd ` Set.filter (λ (y',q') . y' = y) (h M (q,x))
    in if card tgts = 1
      then Some (the_elem tgts)
      else None)"
```

As the name `h_obs` indicates, this function is mostly used in the context of observable FSMs. For such FSMs, there is at most one q' satisfying the above condition. That is, `(h_obs M q x y)` returns `(Some q)` if and only if (q, x, y, q') is a transition of M[6].

Section 12.2 of Part IV introduces refinements on data types `fsm_impl` and `fsm` that facilitate storage of mappings for `h` and `h_obs` in order to avoid repeated computations over the entire set of transitions.

8.2 Well-Formed FSMs

The data type `fsm_impl` introduced in the previous section is sufficient to represent the raw components of a finite state machine, but does not in itself enforce any relations between or properties of them. That is, `fsm_impl` contains values that do not satisfy the properties required of FSMs by Definition 2.2.1: First, the initial state should be contained in the set of states. Next, the sets constituting a finite state machine should be finite[7]. Finally, the states, inputs, and outputs occurring in the transitions should be contained in the respective sets of the FSM. These are common requirements on FSMs (see, for example, [28,

[6] Expression `(f ` A)` is equivalent to `(image f A)`. Function `snd` obtains the second element of a pair. Function `the_elem` obtains the single element in a singleton set.

[7] This requirement could have been enforced by using finite sets (type `fset`) instead of possibly infinite sets (type `set`) for these components. After experimenting with this implementation, I have decided not to do so as I found working with regular sets less cumbersome and also better supported with respect to proof search and automated proof methods.

44, 47, 72, 89]). Some authors (e.g. [107]) furthermore require the input and output alphabets to be non-empty, which I do not require here.

In theory file `FSM.thy`, satisfaction of these requirements is checked by predicate `well_formed_fsm`:

```
abbreviation(input) "well_formed_fsm (M :: ('a,'b,'c) fsm_impl)
     ≡ (initial M ∈ states M
      ∧ finite (states M)
      ∧ finite (inputs M)
      ∧ finite (outputs M)
      ∧ finite (transitions M)
      ∧ (∀ t ∈ transitions M . t_source t ∈ states M ∧
                               t_input t ∈ inputs M ∧
                               t_target t ∈ states M ∧
                               t_output t ∈ outputs M))"
```

Here keyword `abbreviation` indicates that `well_formed_fsm M` is merely a syntactic constant that is replaced by the expression on the right of the ≡ symbol. That is, no unfolding of definitions is required here.

Using this predicate, I then introduce the type for FSMs used throughout the reminder of the formalisation, namely the type `fsm` containing all `fsm_impl` values that satisfy `well_formed_fsm`. This is performed using keyword `typedef`, which introduces a new type by constraining an existing type (for further details, see [119]):

```
typedef ('state, 'input, 'output) fsm =
  "{ M :: ('state, 'input, 'output) fsm_impl . well_formed_fsm M }"
```

As types must be inhabited in Isabelle/HOL, the `typedef` declaration must include a proof that a well-formed FSM exists. This is easily be shown by providing an FSM with a single state and no inputs, outputs, or transitions, which is trivially well-formed. In Isabelle/HOL, this proof may be implemented in a human-readable style, using the following Isar proof (see [121]).

```
proof -
  obtain q :: 'state where "True" by blast
  have "well_formed_fsm (FSMI q {q} {} {} {})" by auto
  then show ?thesis by blast
qed
```

The keyword `obtain` employed in the second line serves to implement proof steps such as *"Let x be an object of type t such that x satisfied property P"*, which may be written in Isar as

```
obtain x :: 't where "P x"
```

As the existence of such an x has to be shown, this introduces the proof goal of showing $(\exists x.\, P(x))$. In the above proof, I merely require q to be some arbitrary state and hence use the trivially true property `True` for `(P x)`. Since types in Isabelle/HOL must be inhabited, the existence of such a q may easily established

by proof method `blast`[8]. Using state q, it is then possible to represent an `fsm_impl` whose initial and only state is q and who does not contain inputs, outputs or transitions, namely `FSMI q {q} {} {} {}`. In the third line of above proof, keyword `have` is used to introduce the fact that this `fsm_impl` is well-formed, which is automatically proven by method `auto`. Finally, the fourth line employs `then` to use the previously established fact in the proof of `?thesis`, which is a shortcut for the goal

$$\exists x.\, x \in \{M \mid \texttt{well\_formed\_fsm}\ M\}$$

This result can again be established by proof method `blast`. Note here that the above proof is more elaborate than strictly necessary, in order to present a first glimpse at several Isar keywords.

In the remainder of this section, I present key aspects of the formalisation provided in `FSM.thy`, including further small examples of proofs as well as various definitions. While numerous (see Table 7.1), the lemmata established in `FSM.thy` are mostly very basic properties of FSMs and can often be proven entirely via automatic proof methods available in Isabelle/HOL. For example, there are more than 20 lemmata on the splitting and combining of paths and the states they reach, of which only 5 require more than a single line of proof. Table 7.2 provides further details on the degree of automation achieved. For an overview of the relevant functions defined in `FSM.thy` see Appendix D.2.

8.2.1 Lifting

Function definitions over type `fsm_impl` may be lifted to type `fsm` via the *Lifting* and *Transfer* packages introduced in [51]. For example, the function `fsm_impl_from_list` defined in the previous section may be lifted in order to obtain an `fsm` instead of an `fsm_impl` from a list of transitions.

```
lift_definition fsm_from_list :: "'a ⇒
                                  ('a,'b,'c) transition list ⇒
                                  ('a,'b,'c) fsm"
  is FSM_Impl.fsm_impl_from_list
```

Obviously, not every function may simply be lifted in this way, as not every value of type `fsm_impl` is well-formed. Thus, in order to lift a function, it must be proven that if it generates only well-formed FSMs. In the following proof, this is performed automatically via structural induction on the list argument, which is given a concrete name *ts* via keyword `fix`:

```
proof -
  fix q  :: 'a
  fix ts :: "('a,'b,'c) transition list"
  show "well_formed_fsm (fsm_impl_from_list q ts)"
    by (induction ts; auto)
qed
```

[8] See Section 9.4.4 of the reference manual [119] for an overview of available fully automated proof methods.

8.2.2 Paths

As introduced in Definition 2.2.2, the notion of *paths* in an FSM is the underlying concept for the language of an FSM and hence for the language-equivalence conformance relation. I have defined paths as lists of transitions and introduce the inductive predicate `path` to check for a given FSM M and state q whether some list of transitions constitutes a path in M starting at q:

```
inductive path :: "('a,'b,'c) fsm ⇒ 'a ⇒ ('a,'b,'c) path ⇒ bool"
  where
  nil[intro!] : "q ∈ states M ⟹ path M q []" |
  cons[intro!] : "t ∈ transitions M ⟹
                  path M (t_target t) ts ⟹
                  path M (t_source t) (t#ts)"
```

Two cases `nil` and `cons` are considered. The former is the base case of the empty path (represented as the empty list `[]`) and specifies that if q is a state of M, then `[]` is a path in M starting at q. The second case, `cons`, specifies how a path is obtained by prepending a transition to another path: if t is a transition in M and ts is a path in M from the target of t, then `(t#ts)` is a path in M from the source of t (recall that `(t#ts)` is the list obtained by prepending element t to list ts).

Using this definition of paths, many basic properties of paths are proven in `FSM.thy`. For example, the following lemma states that if p is a path in M from q[9], then q must be a state of M:

```
lemma path_begin_state :
  assumes "path M q p"
  shows   "q ∈ states M"
  using assms by (cases; auto)
```

This is proven using the assumption (keyword `using`) by considering via proof method `cases` the two cases in which p may be a path (based on the assumption, Isabelle/HOL here automatically selects the rule based on predicate `path` for the case analysis): If case `nil` applies, then q must be a state of M as this is required for case `nil` to apply. If case `cons` applies, then the first element of p must be a transition t of M such that q is the source of t. By the well-formedness of `fsm` M, the source of t – and hence q – must furthermore be a state of M. Each of these cases is proven automatically by proof method `auto`, which is chained to the case analysis via a semicolon.

As a further example on the ability of Isabelle/HOL to automatically prove similar lemmata with minimal guidance, consider the following two lemmata, which show that if the concatenation of paths p_1 and p_2 (written as `(p1@p2)`) constitutes a path in M from q, then its prefix p_1 constitutes a path in M from q and its suffix p_2 constitutes a path in M from the state reached via p_1 applied to q[10].

[9] This assumption is specified using keyword `assume`.

[10] The state reached from a state q via a list of transitions ts is expressed via function `target` as `(target q ts)`. For an overview of functions defined in `FSM.thy` see Appendic D.2.

```
lemma path_prefix :
  assumes "path M q (p1@p2)"
  shows "path M q p1"
  using assms
  by (induction p1 arbitrary: q; auto; (metis path_begin_state))

lemma path_suffix :
  assumes "path M q (p1@p2)"
  shows "path M (target q p1) p2"
  using assms by (induction p1 arbitrary: q; auto)
```

The introduction of `path` enables formalisation of the notion of reachable states. The following definition introduces a predicate `reachable` that indicates a state q to be reachable in M if and only if there exists a path from the initial state of M that reaches q:

```
definition reachable :: "('a,'b,'c) fsm ⇒ 'a ⇒ bool" where
  "reachable M q = (∃ p . path M (initial M) p ∧
                          target (initial M) p = q)"
```

Here, keyword `definition` provides a further way of introducing functions. It differs from `fun` or `function` mainly in that it does not allow recursion or pattern matching on the inputs. It thus avoids checks concerning pattern completeness and termination (see [61]).

Finally, function *after* introduced in Definition 2.3.2, which specifies the state reached by some IO-trace in an observable FSM, is implemented as follows:

```
fun after :: "('a,'b,'c) fsm ⇒ 'a ⇒ ('b × 'c) list ⇒ 'a" where
  "after M q [] = q" |
  "after M q ((x,y)#io) = after M (the (h_obs M q x y)) io"
```

Recall that `(h_obs M q x y)` returns `Some q'` if and only if q' is the unique state such that `(q,x,y,q')` is a transition in M. Function `the` is a predefined function in Isabelle HOL that extracts the q' from `Some q'` and is `undefined` if applied to a `None`. Thus, `(after M q ((x,y)#io))` is `undefined` if there does not exist exactly one transition in M from q with input x and output y. I abbreviate M-after-α as `(after_initial M` α`)`:

```
abbreviation "after_initial M io ≡ after M (initial M) io"
```

8.2.3 Language

The language of an FSM is derived from the IO-projections of its paths. As paths are represented as lists of transitions, I implement the IO-projection as the mapping of each transition t in a path to the pair containing the input and output of t:

```
abbreviation "p_io p ≡ map (λ t . (t_input t, t_output t)) p"
```

The language of state q of FSM M then is the union of all IO-projections (`p_io p`) for all paths p from q in M:

```
fun LS :: "('a,'b,'c) fsm ⇒ 'a ⇒ ('b × 'c) list set" where
  "LS M q = { p_io p | p . path M q p }"
```

That is, (`LS M q`) is the representation of $\mathcal{L}_M(q)$. I abbreviate the language of the initial state of M, that is, the language of M, as (`L M`):

```
abbreviation "L M ≡ LS M (initial M)"
```

These basic definitions and properties on paths and languages are already sufficient to establish some nontrivial properties. For example, the following lemma shows that two FSMs are language-equivalent if there exists an isomorphism on the reachable states. That is, two FSMs $M_1 = (Q, q_0, \Sigma_I, \Sigma_O, h_1)$ and $(S, s_0, \Sigma_I, \Sigma_O, h_2)$ are language-equivalent if there exists a bijective function f from the reachable states of M_1 to those of M_2 such that $f(q_0) = s_0$ and furthermore for reachable states $q, q' \in Q$ and arbitrary x, y it holds that (q, x, y, q') is a transition of M_1 if and only if $(f(q), x, y, f(q'))$ is a transition of M_2:

```
lemma language_equivalence_from_isomorphism_reachable :
  assumes "bij_betw f (reachable_states M1) (reachable_states M2)"
  and     "f (initial M1) = initial M2"
  and     "⋀ q x y q' . q ∈ reachable_states M1 ⟹
                        q' ∈ reachable_states M1 ⟹
                        (q,x,y,q') ∈ transitions M1
                          ⟷ (f q,x,y,f q') ∈ transitions M2"
  shows "L M1 = L M2"
```

The proof I provide for this lemma in `FSM.thy` essentially reduces to establishing that if $(q_0, x, y, q), \ldots, (q', x', y', q'')$ constitutes a path in M_1 then $(f(q_0), x, y, f(q)), \ldots, (f(q'), x', y', f(q''))$ is a path with the same IO-projection in M_2, while it analogously holds that if $(s_0, x, y, s), \ldots, (s', x', y', s'')$ is a path in M_2 then $(f^{-1}(s_0), x, y, f^{-1}(s)), \ldots, (f^{-1}(s'), x', y', f^{-1}(s''))$ is a path with the same IO-projection in M_1.

8.2.4 Basic Properties

Using type `fsm` and logical connectives predefined in Isabelle/HOL, basic properties of FSMs as discussed in Chapter 2 may be defined in a natural way. For example, I have formalised the notions of Definition 2.2.4 in the following predicates:

```
fun deterministic :: "('a,'b,'c) fsm ⇒ bool" where
  "deterministic M =
    (∀ t1 ∈ transitions M . ∀ t2 ∈ transitions M .
      (t_source t1 = t_source t2 ∧ t_input t1 = t_input t2)
        ⟶ (t_output t1 = t_output t2 ∧ t_target t1 = t_target t2))"
```

```
fun observable :: "('a,'b,'c) fsm ⇒ bool" where
  "observable M =
    (∀ t1 ∈ transitions M . ∀ t2 ∈ transitions M .
      (t_source t1 = t_source t2 ∧ t_input t1 = t_input t2 ∧
       t_output t1 = t_output t2)
        ⟶ t_target t1 = t_target t2)"

fun completely_specified :: "('a,'b,'c) fsm ⇒ bool" where
  "completely_specified M = (∀ q ∈ states M .
                              ∀ x ∈ inputs M .
                               ∃ t ∈ transitions M .
                                t_source t = q ∧ t_input t = x)"

fun minimal :: "('a,'b,'c) fsm ⇒ bool" where
  "minimal M = (∀ q ∈ states M . ∀ q' ∈ states M .
                 q ≠ q' ⟶ LS M q ≠ LS M q')"
```

8.2.5 Distinguishing Traces

While not yet describing strategies of their computation, theory file `FSM.thy` already formalises the notion of distinguishing traces in accordance with Definition 2.4.1:

```
definition distinguishes :: "('a,'b,'c) fsm ⇒ 'a ⇒ 'a ⇒
                             ('b ×'c) list ⇒ bool" where
  "distinguishes M q1 q2 io =
    (io ∈ LS M q1 ∪ LS M q2 ∧ io ∉ LS M q1 ∩ LS M q2)"
```

This is extended to the notion of *minimal* distinguishing traces, which are shortest possible distinguishing traces of a given state pair:

```
definition minimally_distinguishes :: "('a,'b,'c) fsm ⇒ 'a ⇒ 'a ⇒
                                       ('b ×'c) list ⇒ bool" where
  "minimally_distinguishes M q1 q2 io =
    (distinguishes M q1 q2 io ∧
      (∀ io' . distinguishes M q1 q2 io' ⟶ length io ≤ length io'))"
```

Minimal distinguishing traces are employed in several proofs, as for example that of Lemma 4.2.5 (further discussed in Subsection 9.1.2). To avoid duplication of proof steps, the following lemma is introduced, which obtains a minimal distinguishing trace for any pair of states q_1, q_2 with distinct languages in M:

```
lemma minimally_distinguishes_ex :
  assumes "LS M q1 ≠ LS M q2"
obtains v where "minimally_distinguishes M q1 q2 v"
proof -
  let ?vs = "{v . distinguishes M q1 q2 v}"
  define vMin where "vMin = arg_min length (λv . v ∈ ?vs)"
  obtain v' where "distinguishes M q1 q2 v'"
    using assms unfolding distinguishes_def by blast
```

```
  then have "vMin ∈ ?vs ∧
             (∀ v'' . distinguishes M q1 q2 v'' ⟶
               length vMin ≤ length v'')"
    unfolding vMin_def
    using arg_min_nat_lemma[of "λv . distinguishes M q1 q2 v" v' length]
    by simp
  then show ?thesis
    using that[of vMin] unfolding minimally_distinguishes_def by blast
qed
```

This proof showcases several Isar keywords not encountered in previously discussed proofs. The first step of the proof (that is, the first line following keyword `proof`) via `let` introduces abbreviation `?vs` for the set of all distinguishing traces of q_1 and q_2. Next, `vMin` is defined as an element of `?vs` of minimal length via pre-defined function `arg_min`. More precisely, `vMin` is a value that satisfies predicate `(λ v. v ∈ ?vs)` such that no other value also satisfying this predicate has a smaller `length` than `vMin`. However, since `?vs` may at this point in the proof still be empty, it is not possible to retrieve these facts on `vMin` yet. Thus, the next line obtains an arbitrary distinguishing trace v' using the assumption of the lemma and unfolding the definition of predicate `distinguishes`. Since the languages differ, there must exist some trace contained in the language of exactly one of the states. By construction, this v' satisfies the containment predicate `(λ v. v ∈ ?vs)` used in the definition of `vMin`. As `length` evaluates to `nat` and type `nat` has a minimal value, namely 0, every non-empty set of traces has a trace of minimal length. This argument is available in the pre-defined lemma `arg_min_nat_lemma`, which is now invoked to make available the properties of `vMin`. Finally, these properties suffice to show that `vMin` is a minimally distinguishing trace.

Minimal distinguishing traces are also employed in Lemma 4.2.4, which I have formalised in Isabelle/HOL as follows:

```
lemma minimally_distinguishes_proper_prefixes_card :
  assumes "observable M"
  and     "minimal M"
  and     "q1 ∈ states M"
  and     "q2 ∈ states M"
  and     "minimally_distinguishes M q1 q2 w"
  and     "S ⊆ states M"
shows "card {w' . w' ∈ set (prefixes w) ∧ w' ≠ w ∧
               after M q1 w' ∈ S ∧ after M q2 w' ∈ S}
       ≤ card S - 1"
```

The assumptions of this `lemma` are immediately recognisable as identical to those of Lemma 4.2.4[11]. Compare next the conclusion of this `lemma` (that is, the part

[11] Following the presentation of Lemma 8.8 of [106], Lemma 4.2.4 furthermore assumes the set S to be non-empty. The proof given in the Isabelle/HOL formalisation does not require this assumption, as for empty S, expression `(card S - 1)` evaluates to 0, since for type `nat` it holds by definition that $0 - 1 = 0$.

after `show`) with that of Lemma 4.2.4:

$$|\{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S \wedge q'\text{-after-}w' \in S\}| \leq |S| - 1$$

The conclusions effectively differ only syntactically, as the formalisation in Isabelle/HOL uses predefined function `card` to obtain the cardinality of a given set[12] and expresses $(w' \in pref(w) \setminus \{w\})$ via the equivalent expression `(w' ∈ set (prefixes w) ∧ w' ≠ w)`[13].

Proof. The mechanised proof I provide for this lemma is a generalisation of the proof given by Soucha in [106] to minimal and possibly partial OFSMs and performs an induction on the size of S. For $|S| = 0$, both calls to `card` in the conclusion return 0 of type `nat`. Here $0 - 1 = 0$ holds for `nat` in Isabelle/HOL, simplifying the conclusion to the trivially true $0 \leq 0$.

In the induction step, let $|S| = k + 1$ for some $k \in \mathbb{N}$ and suppose that the lemma holds for all subsets S' of the states of M of size at most k. Let $w = w'.w''$ where w' denotes the maximum length proper prefix of w such that $q_1\text{-after-}w', q_2\text{-after-}w' \in S$. If no such w' exists, the conclusions follows analogously to the $|S| = 0$ case.

Next, partition S into disjoint sets S_1, S_2 where S_1 denotes the set of states $q \in S$ such that $w'' \in \mathcal{L}_M(q)$ and $S_2 := S \setminus S_1$. Note that $k = |S| = |S_1| + |S_2|$, $S = S_1 \cup S_2$, and $S_1 \cap S_2 = \emptyset$ hold. As $w'.w''$ minimally distinguishes q_1 and q_2, w'' minimally distinguishes $q_1\text{-after-}w'$ and $q_2\text{-after-}w'$, and therefore both S_1 and S_2 are non-empty. Hence, $|S_1| < k$ and $|S_2| < k$ hold, allowing the application of the induction hypothesis to S_1 and S_2.

Partition $W := \{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S \wedge q'\text{-after-}w' \in S\}$ into three sets

$$\begin{aligned}
W_1 &:= \{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S_1 \wedge q'\text{-after-}w' \in S_1\} \\
W_2 &:= \{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S_2 \wedge q'\text{-after-}w' \in S_2\} \\
W' &:= \{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S_1 \wedge q'\text{-after-}w' \in S_2\} \\
&\quad \cup \{w' \mid w' \in pref(w) \setminus \{w\} \wedge q\text{-after-}w' \in S_2 \wedge q'\text{-after-}w' \in S_1\}
\end{aligned}$$

and note that, by definition of S_1,S_2, minimality of w, and maximality of w', if follows that $W' = \{w'\}$. Applying the induction hypothesis to S_1 and S_2 additionally yields $|W_1| \leq |S_1| - 1$ and $|W_2| \leq |S_2| - 1$, which imply the following:

$$|W| = |W_1| + |W_2| + |W'| \leq (|S_1| - 1) + (|S_2| - 1) + 1 = |S_1| + |S_2| - 1 = |S| - 1$$

This provides the desired inequality. □

[12] For infinite sets, `card` returns 0. As the states of M and the set of prefixes of w are finite, this is not relevant for the sets considered in the above lemma.

[13] Function `prefixes` returns a `list`, hence the additional use of `set`.

8.3 State Covers

Theory file `State_Cover.thy` formalises state covers as sets of lists of IO-pairs (that is, as sets of IO-traces) via a type synonym. Predicate `is_state_cover` then checks whether a given set SC constitutes a state cover of FSM M as defined in Definition 2.3.3:

```
type_synonym ('b,'c) state_cover = "('b×'c) list set"

fun is_state_cover :: "('a,'b,'c) fsm ⇒ ('b,'c) state_cover ⇒ bool"
  where
  "is_state_cover M SC =
    ([] ∈ SC ∧ (∀ q ∈ reachable_states M . ∃ io ∈ SC .
                   q ∈ io_targets M io (initial M)))"
```

Here, function application `(io_targets M io q)`, defined in `FSM.thy`, collects all states reachable in M from state q via paths whose IO-projection is io. Thus, `(is_state_cover M SC)` holds if SC contains the empty trace `[]`, reaching the initial state, and furthermore contains for every reachable state q of M an IO-trace reaching that state.

State covers of type `state_cover` do not carry information on which contained traces serve to reach which state of an FSM. Since many algorithms and proofs require this information, I introduce type `state_cover_assignment`, which describes functions from states to IO-traces:

```
type_synonym ('a,'b,'c) state_cover_assignment = "'a ⇒ ('b×'c) list"
```

A given function f then is a state cover assignment of M if it assigns the empty trace to the initial state of M and assigns to each reachable state q of M a trace that reaches q.

```
fun is_state_cover_assignment ::
  "('a,'b,'c) fsm ⇒ ('a,'b,'c) state_cover_assignment ⇒ bool"
  where
  "is_state_cover_assignment M f =
    (f (initial M) = [] ∧
     (∀ q ∈ reachable_states M . q ∈ io_targets M (f q) (initial M)))"
```

By this predicate, applying a state cover assignment to all reachable states of an FSM results in a state cover containing exactly one trace for each reachable state. Thus, state cover assignment alone cannot represent non-minimal state covers. In subsequent definitions and lemmata, I usually employ name V for state cover assignments.

The main objective of theory file `State_Cover.thy` has been the verified computation of state cover assignments. For this purpose, I have implemented a simple breadth-first-search based algorithm for assigning reaching paths to reachable states in function `reaching_paths_up_to_depth`. To simplify the presentation, I provide a shortened pseudocode implementation of this function in Algorithm 39. The function uses five arguments: FSM M, a set nx of states

of M constituting the starting points for the breadth-first-search, a set *done* of states already visited during the search, a partial assignment V mapping states to paths reaching them, and some $k \in \mathbb{N}$ specifying the maximum depth to consider in the search. In each of the k iterations of the loop in Algorithm 39, all transitions (q, x, y, q') of M are considered that emanate from states in nx and that do not target states already in nx or *done*. For each such transition, V is updated such that the path assigned to target q' is the path assigned to q appended with transition (q, x, y, q'). Thus, if $(V\ q)$ is a path reaching q in M, the updated value of $(V\ q')$ is a path to q'.

Note that V as returned by `reaching_paths_up_to_depth` is a map, which is represented in Isabelle/HOL as a function returning values of type `option`. That is, if state q is assigned some path p in V, then `(V q) = Some p` holds. If q is not assigned any value in V, then `(V q) = None` holds.

Algorithm 39: `reaching_paths_up_to_depth`$(M,\ nx,\ done,\ V,\ k)$

$i \leftarrow 0$
while $i < k$ **do**
  $ts \leftarrow \{(q, x, y, q') \in$ `transitions` $M \mid q \in nx \wedge q' \notin nx \cup done\}$
  for $(q, x, y, q') \in ts$ **do**
    update V such that $(V\ q') := (V\ q).(q, x, y, q')$
  $nx \leftarrow \{$`t_target` $t \mid t \in ts\}$
  $done \leftarrow done \cup nx$
  $i \leftarrow i + 1$
return V

I then employ this algorithm in function `get_state_cover_assignment`, which supplies to `reaching_paths_up_to_depth` the parameters required to ensure that the returned assignment of paths to states assigns to each reachable state a path reaching it. From this, a state cover assignment is obtained by replacing the assigned paths with their IO-projections.

```
fun get_state_cover_assignment ::
  "('a::linorder,'b::linorder,'c::linorder) fsm ⇒
  ('a,'b,'c) state_cover_assignment"
  where
  "get_state_cover_assignment M = (let
      q0 = initial M;
      fp = reaching_paths_up_to_depth M {q0} {} [q0↦[]] (size M-1)
    in (λ q . case fp q of Some p ⇒ p_io p | None ⇒ []))"
```

More precisely, `reaching_paths_up_to_depth` is called to start search from the initial state only (nx = `{initial M}`), with no previously handled states ($done$ = `{}`) and having assigned the empty path to the initial state. Since any reachable state is reachable by a path that does not visit any state twice, searching up to depth $k = |M| - 1$ is sufficient to cover all reachable states[14].

[14] The size $|M|$ is computed by `(size M)`, defined in `FSM.thy`.

Thus, this function constitutes an implementation of GETSTATECOVERBYBFS (Algorithm 18) adapted to state cover assignments.

Note here that the mapping returned by `get_state_cover_assignment` returns the empty trace `[]` for any q to which the assignment obtained via `reaching_paths_up_to_depth` does not assign a path. For all other states, the IO-projection `(p_io p)` of assigned path p is returned.

Also note here that function `get_state_cover_assignment` is defined only on `('a,'b,'c) fsm` where the types `'a`, `'b`, `'c` are all of *sort* (or *type class*[15]) `linorder`. That is, the function may only be applied to FSMs whose states, inputs and outputs are of types on whose values a default linear order has been defined, such as $\leq$ on `nat` or `int`. This is required due to some internal steps of `reaching_paths_up_to_depth`, which suppose an order in which to consider the transitions of a given FSM. This is not a serious restriction, as the finiteness of component sets of a well-formed FSM implies enumerability of their elements, allowing for the renaming of these elements with natural numbers. For example, `FSM.thy` provides function `rename_states` that may be employed for this purpose (see Appendix D.2).

The following lemma constitutes the main result of `State_Cover.thy`, showing that `get_state_cover_assignment` does compute a state cover assignment.

```
lemma get_state_cover_assignment_is_state_cover_assignment :
  "is_state_cover_assignment M (get_state_cover_assignment M)"
```

The proof of this lemma consists of two steps. First, it is shown that function `reaching_paths_up_to_depth` applied to the arguments provided to it in `get_state_cover_assignment` returns an assignment fp of paths to states such that fp assigns some p to each q reachable in at most $|M| - 1$ transitions, and that if fp assigns some p to q, then p is a path in M reaching q[16]. Second, it is shown that all reachable states of M are reachable in at most $|M| - 1$ transitions[17].

8.4 Observability Transformation

Theory file `Observability.thy` presents a classical algorithm for transforming an arbitrary FSM into a language-equivalent observable FSM. Consider an FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ and let $M' = (\mathbb{P}(Q), \{q_0\}, \Sigma_I, \Sigma_O, h')$ where

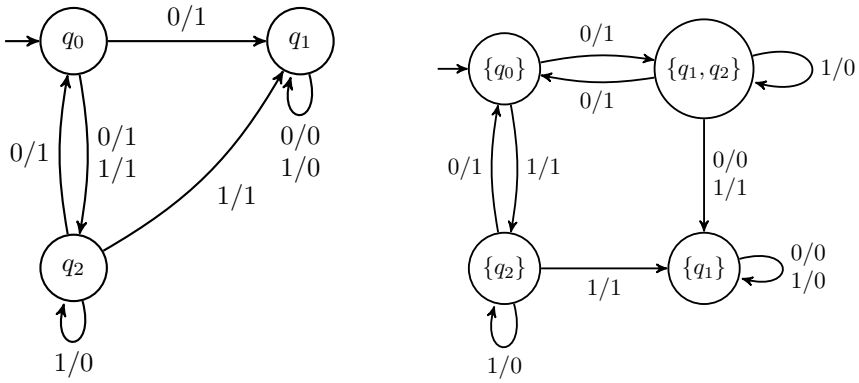
$$h' := \left\{ (A, x, y, \{q' \mid \exists q \in A.\, (q, x, y, q') \in h\}) \;\middle|\; A \subseteq Q \right\}$$

Then it can be proven via induction that there exists a path from A to B in M' if and only if there exists a path with the same IO-projection in M from some

[15] For the development of type classes in Isabelle see [75, 122]. A tutorial on their use with current Isabelle distributions is given in [40].

[16] A generalised version of this result is available in `State_Cover.thy` as lemma `reaching_paths_up_to_depth_set`.

[17] This result is available in `FSM.thy` as lemma `reachable_k_states`.



(a) FSM $M_{8.4}$.

(b) FSM $M'_{8.4}$.

Figure 8.2: Non-observable FSM $M_{8.4}$ and language-equivalent OFSM $M'_{8.4}$.

state $q \in A$ to some $q' \in B$. Hence, as initial state q_0 of M is the only state in the initial state $\{q_0\}$ of M', M and M' are language-equivalent. Furthermore, M' is observable by construction, as the target for any A, x, y is uniquely determined, if it exists. However, this construction of M' always results in an exponential increase in the number of state, of which a large number may not be reachable.

In order to avoid the effort of completing this power set construction, I have chosen to formalise a classical algorithm originally presented in Appendix II of [71], which enumerates only the reachable states of M', thus avoiding superfluous computations on the potentially very large number of unreachable states of M'. For a given reachable state A of M', this algorithm creates the transitions emanating from A by collecting for each IO-pair x/y all transitions of M labelled with that IO-pair whose source lies in A. The union of the targets of these transitions then constitutes a state B in M', and (A, x, y, B) is added to the transitions of M'. This process begins with the initial state $\{q_0\}$ of M' and continues until the emanating transitions of all reachable states have been considered.

Consider FSM $M_{8.4}$ presented in Fig. 8.2a and observable FSM $M'_{8.4}$ obtained by applying the above process to $M_{8.4}$. The initial state of $M'_{8.4}$ is the set $\{q_0\}$ containing only the initial state of $M_{8.4}$. In $M_{8.4}$, the exist three transitions emanating from q_0: one labelled with IO-pair $1/1$ reaching q_2 and two transitions labelled $0/1$ reaching q_1 and q_2. Thus, in $M'_{8.4}$ there exist two transitions emanating from $\{q_0\}$, namely $(\{q_0\}, 1, 1, \{q_2\})$ and $(\{q_0\}, 0, 1, \{q_1, q_2\})$, introducing two new states of $M'_{8.4}$ as targets. Consider next the state $\{q_1, q_2\}$. Transitions emanating from this state are created by combining transitions emanating in $M_{8.4}$ from q_1 or q_2. For example, both states exhibit a self-loop with label $1/0$ and hence $M'_{8.4}$ contains transition $(\{q_1, q_2\}, 1, 0, \{q_1, q_2\})$. Furthermore, q_1 and q_2 reach q_1 via $0/0$ and $1/1$, respectively, introducing transi-

tions $(\{q_1, q_2\}, 0, 0, \{q_1\})$ and $(\{q_1, q_2\}, 1, 1, \{q_1\})$. Note here that the fact that $0/0 \notin \mathcal{L}_{M_{8.4}}(q_2)$ and $1/1 \notin \mathcal{L}_{M_{8.4}}(q_1)$ does not prevent these transitions from being added to $M'_{8.4}$. These transitions introduce a further state $\{q_1\}$ of $M'_{8.4}$. After handling $\{q_2\}$ and $\{q_1\}$, the construction of $M'_{8.4}$ is complete. The resulting FSM exhibits the transitions and states depicted in Fig. 8.2b.

I have implemented the iterative construction process of an observable FSM M' from a given FSM M as a breadth-first-search on the states of M' in function `make_observable_transitions`, which for a set ts_{base} of transitions of M, set nx of states of M' to explore in the next iteration, set $done$ of already explored states of M' and partial transition relation ts_{obs} of M' returns the completed transition relation. Similarly to `reaching_paths_up_to_depth` in the previous section, I present in Algorithm 40 only a simplified pseudocode representation of this function[18].

Algorithm 40: `make_observable_transitions`$(ts_{base}, nx, done, ts_{obs})$

```
while nx ≠ ∅ do
    nx' ← ∅
    for A ∈ nx do
        for x, y such that ∃q, q'. (q, x, y, q') ∈ ts_base ∧ q ∈ A do
            B ← {q' | ∃q. (q, x, y, q') ∈ ts_base ∧ q ∈ A}
            ts_obs ← ts_obs ∪ {(A, x, y, B)}
            nx' ← nx' ∪ {B}
    done ← done ∪ nx
    nx ← nx' \ done
return ts_obs
```

Function `make_observable_transitions` is called by the main function of `Observability.thy`, `make_observable`, which supplies to the former the transitions of M, as well as the initial state of M' and its emanating transitions. Using the list of transitions obtained by this call, `make_observable` then constructs M' by fixing its initial state, setting the input and output alphabets to those of M and finally using as state set the all states occurring in the obtained transitions.

The following lemma verifies the correctness of `make_observable`, that is, its ability to obtain for a given FSM M a language-equivalent FSM M' that is observable and retains the alphabets of M.

```
lemma make_observable_language_observable :
  shows "L (make_observable M) = L M"
    and "observable (make_observable M)"
    and "initial (make_observable M) = {|initial M|}"
```

[18] The concrete implementation of `make_observable_transitions` in `Observability.thy` is less readable without further explanation due to the use of various functions on type `fset`, which I employ instead of `set` to represent sets of states of M.

```
and "inputs (make_observable M) = inputs M"
and "outputs (make_observable M) = outputs M"
```

8.5 Minimisation

In theory file `Minimisation.thy`, I formalise an algorithm to transform an observable FSM into a language-equivalent minimal observable FSM. This is based on a generalisation of *OFSM-Tables* as introduced in [82]. OFSM-Tables are in turn a generalisation of so-called P_k-Tables (see [37]) to observable FSMs.

8.5.1 OFSM-Tables

For a given observable FSM M, the k-th OFSM-Table serves to partition the states of M into classes of states whose languages coincide up to length k. That is, in the 0-th table, all states reside in a single class, while in the first table states q and q' of M reside in the same class if and only if for every IO-pair x/y it holds that x/y is contained either in the language of both states or in neither language.

OFSM-Tables are constructed iteratively, trying to further partition classes by finding IO-pairs which lead pairs of states in the same class to states that reside in different classes in preceding tables. More precisely, the $(k+1)$-th table is derived from the k-th table by creating a row for each state q of M and column for each IO-pair x/y, entering in the corresponding cell either `'-'` if $x/y \notin \mathcal{L}_M(q)$, or otherwise the class that q-after-x/y resides in in the k-th table. States q, q' then reside in the same class in the $(k+1)$-th table if and only if they do so in the k-th table and additionally their rows coincide.

Consider, for example, FSM $M_{8.5}$ depicted in Fig. 8.3. Table 8.1a provides a tabular representation of the transition relation of $M_{8.5}$, specifying for each state q and each IO-pair x/y of $M_{8.5}$ the state reached by that pair from q, if it exists, otherwise using `'-'`. All states reside in the same initial class $c_0 = \{q_0, q_1, q_2, q_3, q_4\}$. Based upon this representation, Table 8.1b shows the first OFSM-Table. Here, all cells not containing `'-'` contain 0, indicating class c_0, as in the 0-th table all states reside in the same class. As states q_0, q_1, q_2 all contain 0/0, 0/1, 1/1 but not 1/0 in their languages, they thus exhibit the same rows in the first OFSM-Table and are grouped into class c_1. The second class is formed by q_3 and q_4, who share the same rows but differ from those of c_1. That is, the states in c_1 are distinguishable via a single input from those in c_2 by any IO-pair for which the rows differ. However, no single input is sufficient to distinguish pairs of states in a single of the two classes.

Next, Table 8.1c shows the second OFSM-Table, where in each cell for state q and IO-pair $x/y \in \mathcal{L}_M(q)$ the class of q-after-x/y in the first OFSM-Table (Table 8.1b) is entered. By this update, the row of state q_2 now differs for IO-pair 0/0 from those of q_0 and q_1 with which it has been grouped together in the first table and which still share the same rows in the second. That is, applying 0/0 to q_0 and q_2, respectively, reaches states that reside in different classes in

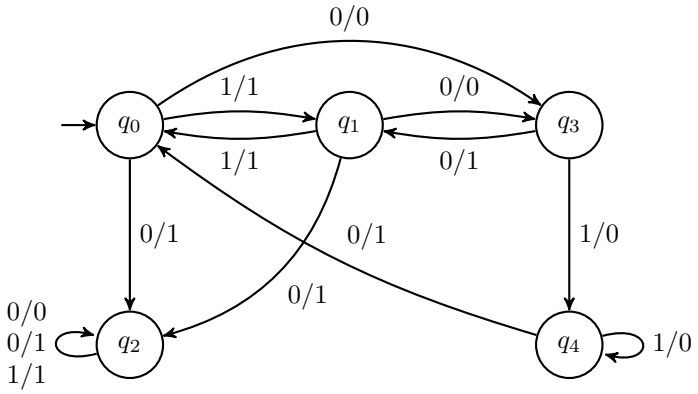


Figure 8.3: FSM $M_{8.5}$.

the first table and hence can be distinguished e.g. by applying 0/0 again. Thus, class c_1 is split into $c_3 = \{q_0, q_1\}$ and $c_4 = \{q_2\}$. States q_3 and q_4 remain in class c_2, as their rows remain identical.

Finally, the third OFSM-Table for $M_{8.5}$ is given in Table 8.1d. Here the partition induced by the classes of the previous table remains unchanged, as the states in $c_3 = \{q_0, q_1\}$ and $c_2 = \{q_3, q_4\}$ respectively again exhibit identical rows. As this table introduces no new classes, all subsequent tables are identical to it. States assigned to the same class in this table are not distinguishable, since if some distinguishing trace ω were to exist for them, then the states would reside in different classes in the $|\omega|$-th OFSM-Table. Thus, in $M_{8.5}$ states q_0 and q_1 exhibit the same language, as do states q_3 and q_4.

I formalise OFSM-Tables as a function `ofsm_table` where call `(ofsm_table M f k q)` returns the class assigned to state q of M in the k-th OFSM-Table. This function generalises the concept of OFSM-Tables presented above by allowing alternative partitions in the 0-th OFSM-Table, which are supplied via procedural parameter f, assigning to each state its class in the 0-th OFSM-Table. That is, two states q, q' of M reside in the same class in `(ofsm_table M f k)` if and only if they are not distinguishable by any trace of length at most k and furthermore each trace $\alpha \in \mathcal{L}_M(q) \cap \mathcal{L}_M(q')$ of length at most k satisfies

$$f(q\text{-after-}\alpha) = f(q'\text{-after-}\alpha)$$

For states q, q' it is assumed that $f(q)$ contains q' if and only if $f(q')$ contains q[19]. Thus, `(ofsm_table M f 0 q)` simply returns $f(q)$. For `(ofsm_table M f (Suc k) q)`, that is, the $(k+1)$-th table, first the class assigned to q in the k-th table `(ofsm_table M f k q)` is computed and then q is assigned the set of all states q' in that class such that for all IO-pairs x/y it holds that either x/y is not in the language of either state or that x/y is in the language of both states and q-after-x/y resides in the same class as q'-after-x/y in the k-th table.

[19] In subsequent sections, f usually assigns to each state the set of all states of M.

Table 8.1: Tabular representation (also representing OFSM-Table 0) and OFSM-Tables 1 to 3 of FSM $M_{8.5}$ using the following classes: $c_0 = \{q_0, q_1, q_2, q_3, q_4\}$, $c_1 = \{q_0, q_1, q_2\}$, $c_2 = \{q_3, q_4\}$, $c_3 = \{q_0, q_1\}$, $c_4 = \{q_2\}$.

(a) Tabular representation of $M_{8.5}$

q_i	0/0	0/1	1/0	1/1	c_i
0	q_3	q_2	-	q_1	0
1	q_3	q_2	-	q_0	0
2	q_2	q_2	-	q_2	0
3	-	q_1	q_4	-	0
4	-	q_0	q_4	-	0

(b) 1st OFSM-Table

q_i	0/0	0/1	1/0	1/1	c_i
0	0	0	-	0	1
1	0	0	-	0	1
2	0	0	-	0	1
3	-	0	0	-	2
4	-	0	0	-	2

(c) 2nd OFSM-Table

q_i	0/0	0/1	1/0	1/1	c_i
0	2	1	-	1	3
1	2	1	-	1	3
2	1	1	-	1	4
3	-	1	2	-	2
4	-	1	2	-	2

(d) 3rd OFSM-Table

q_i	0/0	0/1	1/0	1/1	c_i
0	2	4	-	3	3
1	2	4	-	3	3
2	4	4	-	4	4
3	-	3	2	-	2
4	-	3	2	-	2

This is implemented as follows, using `h_obs` to both check whether x/y is in the language of some state and, if it is, to obtain the state reached by it[20].

```
fun ofsm_table ::
  "('a,'b,'c) fsm ⇒ ('a ⇒ 'a set) ⇒ nat ⇒ 'a ⇒ 'a set"
  where
  "ofsm_table M f 0 q = (if q ∈ states M then f q else {})" |
  "ofsm_table M f (Suc k) q = (let
      prev_table = ofsm_table M f k
    in {q' ∈ prev_table q . ∀ x ∈ inputs M . ∀ y ∈ outputs M .
        (case h_obs M q x y of
          Some qT ⇒ (case h_obs M q' x y of
            Some qT' ⇒ prev_table qT = prev_table qT' |
            None ⇒ False) |
          None ⇒ h_obs M q' x y = None) })"
```

Note that this algorithm, while intuitive and easy to use in proofs, is not very efficient, as it performs possibly many separate recursive calls in each invocation and does not store intermediate results. Furthermore, later applications require the classes of all states of an FSM, which further repeats computations due to the lack of storing intermediate tables. Theory file `OFSM_Tables_Refined.thy` discussed in Section 12.3 introduces functions for a more efficient way of com-

[20] As OFSM-Tables are only suitable to be applied to observable FSMs, this does not constitute a restriction.

puting the required tables and storing them as maps instead of functions. Due to the use of *code equations* (see Chapter 12), proofs on algorithms described in the remainder of this section and also in Section 8.6, which also rely on OFSM-Tables, need not be modified or repeated to make use of these more efficient implementations.

8.5.2 A Minimisation Algorithm

An observable FSM M may be transformed into a language-equivalent minimal observable FSM M' by conflating language-equivalent states. Let $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$. For each $q \in Q$, let $[q] := \{q' \in Q \mid \mathcal{L}_M(q) = \mathcal{L}_M(q')\}$ denote the set of all states of M that exhibit the same language as q. Note that $\{[q] \mid q \in Q\}$ constitutes a partition of Q and that $\{(q, q') \mid q, q' \in Q \wedge q' \in [q]\}$ constitutes an equivalence relation. Using the equivalence classes $[q]$ instead of q for each state of M, M' is constructed as

$$M' := (\{[q] \mid q \in Q\}, [q_0], \Sigma_I, \Sigma_O, h'), \text{ where}$$
$$h' := \{([q], x, y, [q']) \mid (q, x, y, q') \in h\}$$

This construction is an adaption of the classical algorithm for minimisation of deterministic finite automata (DFA) (see, for example, [46]). Here, an observable FSM is effectively interpreted as a DFA with the same set of states whose transitions are obtained by interpreting each transition (q, x, y, q') of the FSM as a transition $(q, (x, y), q')$. That is, the IO-pairs of the FSM are considered as symbols in the input alphabet of the DFA, where observability of the FSM ensures determinism of the automaton.

Class $[q]$ for each $q \in Q$ may be computed via OFSM-Tables. More precisely, language-equivalent states may be identified by finding the first OFSM-Table such that subsequent tables do not change the partition, as the classes induced by this table are the desired equivalence classes. I implement the search for this "last" OFSM-Table in function `(ofsm_table_fix M f k)`, which returns `(ofsm_table M f n)` for the smallest $n \geq k$ such that for all states q the classes assigned in `(ofsm_table M f n)` and `(ofsm_table M f (Suc n))` coincide[21]. That is, `(ofsm_table_fix M f 0 q)` returns [q] if f assigns to each state $q \in Q$ the set Q.

```
function ofsm_table_fix ::
  "('a,'b,'c) fsm ⇒ ('a ⇒ 'a set) ⇒ nat ⇒ 'a ⇒ 'a set"
  where
  "ofsm_table_fix M f k = (let
    cur_table = ofsm_table M (λq. f q ∩ states M) k;
    next_table = ofsm_table M (λq. f q ∩ states M) (Suc k)
```

[21] More precisely, `(ofsm_table_fix M f k)` passes to the recursive calls as procedural parameter a function derived from f by restricting $f(q)$ for each q to a subset of the states of M. I have added this restriction to simplify the termination proof by establishing finiteness of the classes assigned by f. For the values of f passed to `ofsm_table_fix` in the present work, this poses no restriction.

```
  in if (∀ q ∈ states M . cur_table q = next_table q)
    then cur_table
    else ofsm_table_fix M f (Suc k))"
  by pat_completeness auto
```

I include the implementation here for the purpose of showcasing a function where keyword `fun` is not sufficient to define the function, as the termination criterion for trying values $k, \ldots, n$ is not obvious to the applied automated proof methods of Isabelle/HOL. Thus, termination needs to be proven manually, following keyword `termination`. To this purpose, I employ the simple measure of counting the combined sizes of the classes assigned to each state of M, without any special handling of duplicates:

```
termination
  apply (relation "measure (λ (M, f, k) . ∑ q ∈ states M .
                    card (ofsm_table M (λq. f q ∩ states M) k q))")
```

This measure decreases in the recursive calls, as these only occur if the k-th and $(k+1)$-th table differ, which is only possible if some states q_1, q_2 were assigned to the same class c in the k-th table and have been assigned new classes $c_1, c_2 \subsetneq c$. Thus, the sizes of classes assigned to q_1 and q_2 must decrease. Furthermore, no state is assigned a larger class in the $(k+1)$-th table than in the k-th table. I here omit the formalisation of this argument, which follows after above listing in `Minimisation.thy`.

Finally, the construction of language-equivalent minimal observable FSM M' from observable FSM M as described in the beginning of this subsection is implement in function `minimise`, using `ofsm_table_fix` to compute equivalence classes. Evaluation of `(minimise M)` thus maps every transition (q, x, y, q') of M to $([q], x, y, [q'])$ and sets the initial state of the returned FSM to `[initial M]`. Correctness of this implementation is established via the following lemmata, which show that if M is observable, then `(minimise M)` is observable, minimal, and language-equivalent to M.

```
lemma minimise_observable:
  assumes "observable M"
  shows "observable (minimise M)"

lemma minimise_minimal:
  assumes "observable M"
  shows "minimal (minimise M)"

lemma minimise_language:
  assumes "observable M"
  shows "L (minimise M) = L M"
```

The first of these lemmata follows immediately from the construction of M' and the observability of M. The remaining two lemmata follow from establishing that $\mathcal{L}_M(q) = \mathcal{L}_{M'}([q])$ and the fact that if $[q] \neq [q']$, then $\mathcal{L}_M(q) \neq \mathcal{L}_M(q')$.

As detailed in the previous subsection and Table 8.1d, for FSM $M_{8.5}$ depicted in Fig. 8.3, there exist three classes of language-equivalent states: $c_3 = \{q_0, q_1\}$,

$c_4 = \{q_2\}$, and $c_2 = \{q_3, q_4\}$. That is, $[q_0] = [q_1] = c_3$, $[q_2] = c_4$, and $[q_3] = [q_4] = c_2$. Thus, minimisation of $M_{8.5}$ returns FSM $M'_{8.5}$ depicted in Fig. 8.4.

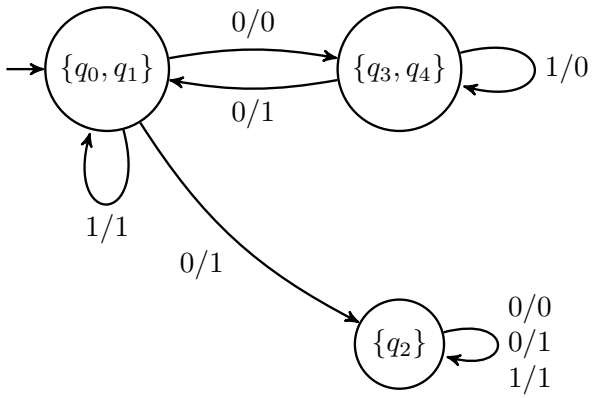


Figure 8.4: Minimisation $M'_{8.5}$ of FSM $M_{8.5}$ of Fig. 8.3.

8.6 Computation of Distinguishing Traces

The OFSM-Tables formalised in the previous section are of use not only in minimisation but also in computing distinguishing traces of minimal length. Suppose that states q, q' of observable FSM M reside in the same class in the k-th table but are assigned distinct classes in the $(k+1)$-th table. This is only possible if the rows for q and q' differ in the latter table. That is, if there exists some IO-pair x/y such that either $k = 1$ and x/y distinguishes q and q', or if $k > 1$, $x/y \in \mathcal{L}_M(q) \cap \mathcal{L}_M(q')$, and the states q-after-x/y and q'-after-x/y reside in distinct classes in the k-th table. In the latter case, x/y is the first IO-pair of a distinguishing trace whose subsequent pairs are obtained by repeating the same process on q-after-x/y and q'-after-x/y. Note here that q-after-x/y and q'-after-x/y must reside in the same class in all tables before the k-th table, as otherwise q and q' would already reside in distinct classes in the k-th table.

For example, the second OFSM-Table (Table 8.1c) for FSM $M_{8.5}$ depicted in Fig. 8.3 shows that IO-pair 0/0 leads states q_0 and q_2 to states q_3 and q_2, respectively, which reside in distinct classes in the first OFSM-Table. In turn, the first OFSM-Table (Table 8.1b) shows that IO-pair 1/0 distinguishes q_3 and q_2, as it is contained only in the language of q_3. Thus, $(0/0).(1/0) = 01/00 \in \mathcal{L}_M(q_0) \setminus \mathcal{L}_M(q_2)$ constitutes a distinguishing trace of q_0 and q_2 in $M_{8.5}$.

This process of iteratively selecting IO-pairs leading to states residing in distinct classes of the previous table is implemented in `Distinguishability.thy` as function `assemble_distinguishing_sequence_from_ofsm_table`. The selection of some specific IO-pair as described above is performed via function `select_diverging_ofsm_table_io`.

```
fun assemble_distinguishing_sequence_from_ofsm_table ::
  "('a,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ nat ⇒
    ('b×'c) list"
  where
  "assemble_distinguishing_sequence_from_ofsm_table M q1 q2 0 = []" |
  "assemble_distinguishing_sequence_from_ofsm_table M q1 q2 (Suc k) =
    (case select_diverging_ofsm_table_io M q1 q2 (Suc k) of
      ((x,y),(Some q1',Some q2')) ⇒ (x,y) #
        assemble_distinguishing_sequence_from_ofsm_table M q1' q2' k |
      ((x,y),_) ⇒ [(x,y)])"
```

In order to compute *minimal* distinguishing traces, the index k of the first OFSM-Table in which the states that are to distinguish are assigned distinct classes needs to be supplied. I implement the search for this index in function `find_first_distinct_ofsm_table` and combine the two functions into function `get_distinguishing_sequence_from_ofsm_tables`.

```
fun get_distinguishing_sequence_from_ofsm_tables ::
  "('a,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ ('b × 'c) list"
  where
  "get_distinguishing_sequence_from_ofsm_tables M q1 q2 = (let
      k = find_first_distinct_ofsm_table M q1 q2
  in assemble_distinguishing_sequence_from_ofsm_table M q1 q2 k)"
```

The following lemma establishes that this function, applied to states $q_1 \neq q_2$ of a minimal OFSM M computes a distinguishing trace of minimal length. Its proof is based on the definition of OFSM-Tables and induction on the index k of the first OFSM-Table assigning different classes to q_1 and q_2.

```
lemma
get_distinguishing_sequence_from_ofsm_tables_is_minimally_distinguishing:
  fixes M :: "('a,'b::linorder,'c::linorder) fsm"
  assumes "observable M"
  and     "minimal M"
  and     "q1 ∈ states M"
  and     "q2 ∈ states M"
  and     "q1 ≠ q2"
  shows "minimally_distinguishes M q1 q2
           (get_distinguishing_sequence_from_ofsm_tables M q1 q2)"
```

Chapter 9

Convergence

The establishment of convergence between certain traces in the system under test constitutes the main feature of the SPY and SPYH-Methods (see [103, 108]), allowing these methods to reduce test suite size by distributing separating traces over convergent traces. In this chapter, I introduce the formalisation of various concepts related to convergence. To this end, Section 9.1 introduces the formalised definition of convergence in OFSMs, as well as sufficient conditions on establishing convergence in an OFSM based on some test suite. Next, Section 9.2 defines the requirements on convergence graphs introduced in Subsection 3.3, which provide an interface between concrete implementations of convergence graphs and the various steps in test strategies employing them. Finally, Sections 9.3 and 9.4 introduce two implementations of convergence graphs Empty and Standard as employed in Part 6 and first introduced in Subsection 6.3.4.

Appendices D.7 to D.10 provide an overview of the functions implemented in the theory files described in this chapter.

9.1 Establishing Convergence

In theory file `Convergence.thy`, I employ a definition of convergence that is slightly more general than that given in Definition 2.5.1 by requiring equality of the language of reached states instead of identity of the reached states:

```
fun converge ::
  "('a,'b,'c) fsm ⇒ ('b × 'c) list  ⇒ ('b × 'c) list ⇒ bool"
  where
  "converge M π τ =
     (π ∈ L M ∧ τ ∈ L M ∧
      (LS M (after_initial M π) = LS M (after_initial M τ)))"
```

Thus, `converge` is also applicable to non-minimal OFSMs, allowing the weakening of the assumptions of some lemmata. On minimal OFSMs the definition

coincides with that given in Definition 2.5.1, as two states of a minimal FSM exhibit the same language if and only if they are the same state.

The preservation of convergence or divergence on some set of traces follows Definitions 2.5.3, but is implemented between a given reference model M_1 and a given system under test M_2, instead of between M_1 and a set of FSMs. Thus, `(preserves_divergence M1 M2 A)` holds if and only if A is $\{M_2\}$-convergence-preserving with respect to M_1.

```
fun preserves_convergence ::
  "('a,'b,'c) fsm ⇒ ('d,'b,'c) fsm ⇒ ('b × 'c) list set ⇒ bool"
  where
  "preserves_convergence M1 M2 A =
    (∀ α ∈ L M1 ∩ A . ∀ β ∈ L M1 ∩ A .
        converge M1 α β ⟶ converge M2 α β)"

fun preserves_divergence ::
  "('a,'b,'c) fsm ⇒ ('d,'b,'c) fsm ⇒ ('b × 'c) list set ⇒ bool"
  where
  "preserves_divergence M1 M2 A =
    (∀ α ∈ L M1 ∩ A . ∀ β ∈ L M1 ∩ A .
        ¬ converge M1 α β ⟶ ¬ converge M2 α β)"
```

An implementation directly representing Definition 2.5.3 could thus easily be defined by requiring `(preserves_divergence M1 M2 A)` to hold for all M_2 in a given set of FSMs. However, in subsequent lemmata and definitions I continue to express relations originally defined between reference model M and a set of FSM (usually $\mathcal{F}(M,m)$) as relations between M and a single other FSM. I have chosen this approach, as there is no single class of FSMs suitable for all lemmata. In particular, $\mathcal{F}(M,m)$ is more restrictive than strictly necessary, for example when considering lemmata that do not require minimality or observability of M or the behaviour of the system under test.

9.1.1 Lemma 4.2.3

For example, the original formulation of Lemma 4.2.3 in [106] (a revised form of Lemma 3 of [103]) relates M to some $N \in \mathcal{F}(M,m)$, but does not require all properties implied by containment in $\mathcal{F}(M,m)$ in the proof, such as the upper bound m on size. Furthermore, $\mathcal{F}(M,m)$-divergence-preservation is required on some traces, while the proof requires only $\{N\}$-divergence-preservation. Thus, in Lemma 4.2.3 I relate M (M_1) to an arbitrary N (M_2) of which I assume only minimality and observability instead of $N \in \mathcal{F}(M,m)$, avoiding unnecessary assumptions. The following lemma finally formalises Lemma 4.2.3 in `Convergence.thy`.

```
lemma
  preserves_divergence_minimally_distinguishing_prefixes_lower_bound :
  fixes M1 :: "('a,'b,'c) fsm"
  fixes M2 :: "('d,'b,'c) fsm"
```

```
  assumes "observable M1"
  and     "observable M2"
  and     "minimal M1"
  and     "minimal M2"
  and     "converge M1 u v"
  and     "¬converge M2 u v"
  and     "u ∈ L M2"
  and     "v ∈ L M2"
  and     "minimally_distinguishes M2 (after_initial M2 u)
                                      (after_initial M2 v) w"
  and     "wp ∈ set (prefixes w)"
  and     "wp ≠ w"
  and     "wp ∈ LS M1 (after_initial M1 u) ∩
                LS M1 (after_initial M1 v)"
  and     "preserves_divergence M1 M2
                {α@γ | α γ. α ∈ {u,v} ∧ γ ∈ set (prefixes wp)}"
  and     "L M1 ∩ {α@γ | α γ. α ∈ {u,v} ∧ γ ∈ set (prefixes wp)}
         = L M2 ∩ {α@γ | α γ. α ∈ {u,v} ∧ γ ∈ set (prefixes wp)}"
shows "card (after_initial M2 ‘
                {α@γ | α γ. α ∈ {u,v} ∧ γ ∈ set (prefixes wp)})
       ≥ 1 + length wp + (card (FSM.after M1 (after_initial M1 u) ‘
                                              (set (prefixes wp))))"
```

Compare this to the natural language version given in Lemma 4.2.3, which I repeat below for easier reference, replacing the names employed in the original statement with those used in the Isabelle/HOL formalisation:

Lemma. *Let M_1 and M_2 be minimal OFSMs (assumptions 1-4). Suppose that $u, v \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ converge in M_1 but not in M_2 (assumptions 5-8). Furthermore suppose that $w \in \Delta_{M_2}(M_2\text{-after-}u, M_2\text{-after-}v)$ is a minimal length trace distinguishing the states reached by u,v in M_2 (assumption 9). Let w_p be some proper prefix of w (assumptions 10,11) such that $w_p \in \mathcal{L}_{M_1}(M_1\text{-after-}u) \cap \mathcal{L}_{M_1}(M_1\text{-after-}v)$ (assumption 12) and suppose that $\{u, v\}.pref(w_p)$ is $\{M_2\}$-divergence-preserving (assumption 13[1]). Finally suppose that M_2 passes trace set $\{u, v\}.pref(w_p)$ with respect to M_1 (assumption 14).*

Then the traces in $\{u, v\}.pref(w_p)$ reach at least

$$|w_p| + \left| \bigcup_{w_p' \in pref(w_p)} \{M_1\text{-after-}u.w_p'\} \right| + 1$$

distinct states of M_2 ⊣

I prove the above lemma via a generalisation of the proof originally provided by Soucha in [106] for complete deterministic FSMs, where I follow a proof developed by Wen-ling Huang and provided to me via email on June 28, 2021.

[1] In the Isabelle/HOL formalisation I unfold the set $\{u, v\}.pref(w_p)$ to the equivalent $\{\alpha.\gamma \mid \alpha \in \{u, v\} \wedge \gamma \in pref(w_p)\}$.

Proof. Let $k = |w_p|$ and let w_i denote the prefix of w of length $0 \leq i \leq k$. Note that w_i by construction also is a prefix of w_p. Let q_u, q_v and t_u, t_v denote the states reached by u,v in M_1, M_2 respectively.

As w_p minimally distinguishes t_u and t_v, $u.w_i$ and $v.w_i$ must diverge in M_2 for all $0 \leq i \leq k$. Let a_i, b_i denote the states of M_2 reached by $u.w_i$ and $v.w_i$, respectively.

In M_1, such $u.w_i$ and $v.w_i$ must converge in some q_i due to the convergence of u and v in that FSM. This allows the set $\{0, \ldots, k\}$ to be partitioned into pairwise disjoint, non-empty classes $\{I_1, \ldots, I_l\}$ such that $i, j \in \{0, \ldots, k\}$ are contained in the same class if and only if $q_i = q_j$ holds. Note that all q_i, a_i, b_i are well-defined for $0 \leq i \leq k$ due to $u, v \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$[2] and $w_p \in \mathcal{L}_{M_1}(M_1\text{-after-}u) \cap \mathcal{L}_{M_1}(M_1\text{-after-}v)$. Also note that, as it is a partition, the union of all I_i contains exactly the set $\{0, \ldots, k\}$, and hence has the same cardinality:

$$\sum_{i=1}^{l} |I_i| = k + 1 \tag{9.1}$$

Furthermore, as each I_i contains only indices of prefixes reaching the same state, which is not reached by any other I_j with $j \neq i$, l also denotes the number of distinct states of M_1 visited by prefixes of w_p:

$$\left| \bigcup_{w_p' \in pref(w_p)} \{M_1\text{-after-}u.w_p'\} \right| = \left| \bigcup_{i=0}^{k} \{q_i\} \right| = l \tag{9.2}$$

Next, let $S_i := \bigcup_{j \in I_i} \{a_j, b_j\}$ for all $1 \leq i \leq l$. For all $1 \leq i < j \leq l$ it holds that $S_i \cap S_k = \emptyset$, since $I_i \cap I_j = \emptyset$ following from $i \neq j$, which implies $q_i \neq q_j$ and hence $\{a_i, b_i\} \cap \{a_j, b_j\} = \emptyset$ due to the assumed preservation of convergence in M_2 over $\{u, v\}.pref(w_p)$ with respect to M_1. Therefore, the combined size of the S_i sets may be defined as follows:

$$\left| \bigcup_{i=1}^{l} S_i \right| = \sum_{i=1}^{l} |S_i| \tag{9.3}$$

Additionally, the fact that $\{I_1, \ldots, I_l\}$ partitions $\{0, \ldots, k\}$ implies that the following holds by definition of S_i:

$$\left| \bigcup_{i=1}^{l} S_i \right| = \left| \bigcup_{i=0}^{k} \{a_i, b_i\} \right| \tag{9.4}$$

Combining (9.3) and (9.4) allows describing the number of distinct states of M_2 reached by prefixes of w_p in terms of the S_i sets:

$$\left| \bigcup_{i=0}^{k} \{a_i, b_i\} \right| = \sum_{i=1}^{l} |S_i| \tag{9.5}$$

[2]Recall that convergence in M_1 implies containment in $\mathcal{L}(M_1)$.

Furthermore, Lemma 4.2.4 (formalised in Subsection 8.2.5) may be applied to obtain an upper bound on the size of each S_i for $1 \leq i \leq l$, as S_i is a subset of the states of minimal OFSM M_2 and w minimally distinguishes states t_u, t_v of M_2. That is,

$$|\{w' \mid w' \in pref(w) \setminus \{w\} \wedge t_u\text{-after-}w' \in S_i \wedge t_v\text{-after-}w' \in S_i\}| \leq |S_i| - 1$$

holds, which, by definitions of traces w_j, which are proper prefixes of w, as well as of a_i and b_i may be refined to

$$|\{w_j \mid 0 \leq j \leq k \wedge a_j \in S_i \wedge b_j \in S_i\}| \leq |S_i| - 1 \tag{9.6}$$

By definition of S_i, the set on the left hand side of this inequality contains w_j for all $j \in I_i$. As prefixes of w of different length are trivially distinct, the set on the left hand side thus contains at least $|I_i|$ distinct elements. Hence, the following lower bound on the size of each S_i with $1 \leq i \leq l$ may be derived:

$$|S_i| \geq |I_i| + 1 \tag{9.7}$$

Using the above results, the number of distinct states of M_2 reached by traces in $\{u, v\}.pref(w_p)$, that is, $|\bigcup_{i=0}^{k}\{a_i, b_i\}|$, may finally be limited by the desired lower bound as follows:

$$\begin{aligned}
\left|\bigcup_{i=0}^{k}\{a_i, b_i\}\right| &= \sum_{i=1}^{l}|S_i| && \text{(by (9.5))}\\
&\geq \sum_{i=1}^{l}(|I_i| + 1) && \text{(by (9.7))}\\
&= l + \sum_{i=1}^{l}|I_i| \\
&= k + l + 1 && \text{(by (9.1))}\\
&= |w_p| + \left|\bigcup_{w_p{}' \in pref(w_p)}\{M_1\text{-after-}u.w_p{}'\}\right| + 1 && \text{(by (9.2))}
\end{aligned}$$

□

9.1.2 Lemma 4.2.5

The main result on convergence available in theory file `Convergence.thy` consists in a formalisation of Lemma 4.2.5 and its abstraction Lemma 4.2.6, which describe sufficient conditions for the establishment of convergence of two traces in the system under test. The first of these lemmata is stated in Isabelle/HOL as follows:

```
lemma sufficient_condition_for_convergence :
  fixes M1 :: "('a,'b,'c) fsm"
  fixes M2 :: "('d,'b,'c) fsm"
  assumes "observable M1"
  and     "observable M2"
  and     "minimal M1"
  and     "minimal M2"
  and     "size_r M1 ≤ m"
  and     "size M2 ≤ m"
  and     "inputs M2 = inputs M1"
  and     "outputs M2 = outputs M1"
  and     "converge M1 π τ"
  and     "L M1 ∩ T = L M2 ∩ T"
  and     "⋀ γ x y . length (γ@[(x,y)]) ≤ m - size_r M1 ⟹
                γ ∈ LS M1 (after_initial M1 π) ⟹
                x ∈ inputs M1 ⟹ y ∈ outputs M1 ⟹
                ∃ SC α β . SC ⊆ T
                           ∧ is_state_cover M1 SC
                           ∧ {ω@ω' | ω ω' . ω ∈ {α,β} ∧
                               ω' ∈ set (prefixes (γ@[(x,y)]))} ⊆ SC
                           ∧ converge M1 π α
                           ∧ converge M2 π α
                           ∧ converge M1 τ β
                           ∧ converge M2 τ β
                           ∧ preserves_divergence M1 M2 SC"
  and     "∃ SC α β . SC ⊆ T
                     ∧ is_state_cover M1 SC
                     ∧ α ∈ SC ∧ β ∈ SC
                     ∧ converge M1 π α
                     ∧ converge M2 π α
                     ∧ converge M1 τ β
                     ∧ converge M2 τ β
                     ∧ preserves_divergence M1 M2 SC"
shows "converge M2 π τ"
```

For easier reference, I again copy the natural language description of the lemma (given in Lemma 4.2.5) below, replacing the names employed in the original statement with those used in the Isabelle/HOL formalisation and relating reference model M_1 with some specific M_2 in the fault domain $\mathcal{F}(M_1, m)$:

Lemma. *Let M_1 and M_2 be minimal OFSMs (assumptions 1-4) over the same input and output alphabets (assumptions 7,8) and suppose that M_1 has n reachable states such that $n \leq m$ (assumption 5), while M_2 has at most m states (assumption 6)*[3]*. Suppose that π, τ converge in M_1 (assumption 9) and let TS be a test suite passed by M_2 with respect to M_1 (assumption 10). Furthermore suppose that for all IO-traces γ of length at most $m - n$ over the inputs and outputs of M_1 whose proper prefixes are in the language of M_1-after-π, there*

[3] That is, assumptions 1-8 imply $M_2 \in \mathcal{F}(M_1, m)$.

exist some $\alpha \in [\pi]_{TS}$[4]*,* $\beta \in [\tau]_{TS}$ *and a state cover* $SC \subseteq TS$ *of* M_1 *that contains* $\{\alpha, \beta\}.pref(\gamma)$ *and is* $\{M_2\}$*-divergence-preserving with respect to* M_1 *(assumptions 11,12). Then* π *and* τ *converge in* M_2*.* ⊣

The assumption on coverage of IO-traces γ is encoded in two separate assumptions in the formalisation. Assumption 12 covers case $\gamma = \epsilon$, whereas assumption 11 covers case $\gamma = \gamma'.(x/y)$, ensuring that γ' is in the language of the state of M_1 reached by π (and hence τ) and that x/y constitutes an IO-pair over the alphabets of M_1.

Proof. Similarly to the previous subsection, I prove this lemma using a generalisation of the proof developed by Simão et al. in Theorem 2 of [103], based primarily on Lemma 4.2.3 (see the previous subsection) and Lemma 4.2.4 (see Subsection 8.2.5).

Suppose that π and τ do *not* converge in M_2. Then there must exist a distinguishing trace γ of M_2-after-π and M_2-after-τ. As π and τ converge in M_1, it follows for every γ' that $\pi.\gamma' \in \mathcal{L}(M_1)$ holds if and only if $\tau.\gamma' \in \mathcal{L}(M_1)$ holds. Furthermore, by assumptions 11 and 12, for every prefix γ' of γ of length at most $m - n$ it holds that $\pi.\gamma' \in \mathcal{L}(M_2)$ holds if and only if $\tau.\gamma' \in \mathcal{L}(M_2)$, as otherwise M_2 could not pass TS. Thus, γ must be of length at least $m - n + 1$, as it is a distinguishing trace of M_2-after-π and M_2-after-τ, where $\pi, \tau \in \mathcal{L}(M_2)$ by assumption 12.

Let γ' be the prefix of γ of length $m-n$. By assumptions 11 and 12 (depending on the length of γ'), there exist $\alpha \in [\pi]_{TS}$, $\beta \in [\tau]_{TS}$ and a state cover $SC \subseteq TS$ such that $\{\alpha, \beta\}.pref(\gamma') \subseteq SC$ and SC is $\{M_2\}$-divergence-preserving with respect to M_1. By Lemma 4.2.3, minimality of γ and $|\gamma'| = m - n$, the number of distinct states reached in M_2 via $\{\alpha, \beta\}.pref(\gamma')$ is at least

$$(m - n) + \left| \bigcup_{\gamma'' \in pref(\gamma')} \{M_1\text{-after-}\alpha.\gamma''\} \right| + 1 \tag{9.8}$$

Since SC is a state cover of M_1, reaching all n distinct reachable states of it, and α, β converge in M_1, there exist $n - |\bigcup_{\gamma'' \in pref(\gamma')} M_1\text{-after-}\alpha.\gamma''|$ traces in SC reaching distinct states in M_1 not reached by $\{\alpha, \beta\}.pref(\gamma')$. As the latter is a subset of SC and SC is $\{M_2\}$-divergence-preserving, these traces must also in M_2 reach $n - |\bigcup_{\gamma'' \in pref(\gamma')} M_1\text{-after-}\alpha.\gamma''|$ distinct states not reached via $\{\alpha, \beta\}.pref(\gamma')$. Thus, by (9.8), the traces in $\{\alpha, \beta\}.pref(\gamma')$ must reach $m + 1$ distinct states of M_2, contradicting the assumption that M_2 contains at most m states. □

In theory file `Convergence.thy`, I establish the abstraction of Lemma 4.2.5 to Lemma 4.2.6 in lemma `establish_convergence_from_pass`[5]. This lemma

[4] As M_2 is assumed to pass TS, notation $\alpha \in [\pi]_{TS}$ is unfolded to the conjunction of `(converge M1` π α`)` and `(converge M2` π α`)`.

[5] The Isabelle/HOL formalisation of this lemma superficially differs from the statement of Lemma 4.2.6 in Section 4.2 in that it does not explicitly name test suites TS or TS'.

later is employed instead of the above result in establishing correctness of certain steps of the SPY and H-Frameworks, as it does not require finding or storing witnesses for the various SC, α, and β, simplifying the mechanisation of some proofs.

9.1.3 Completeness from Convergence

I conclude theory file `Convergence.thy` with a formalisation of Corollary 4.2.2, which describes a sufficient condition for m-completeness based on establishing a transition cover with certain properties. This first requires formalising the definition of transition covers following Definition 2.3.4.

```
definition transition_cover ::
  "('a,'b,'c) fsm ⇒ ('b × 'c) list set ⇒ bool"
  where
  "transition_cover M A =
    (∀ q ∈ reachable_states M . ∀ x ∈ inputs M . ∀ y ∈ outputs M .
      ∃ α. α ∈ A ∧ α@[(x,y)] ∈ A ∧ α ∈ L M ∧ after_initial M α = q)"
```

The formalisation of Corollary 4.2.2 may then be stated as follows:

```
lemma initialised_convergence_preserving_transition_cover_is_complete :
  fixes M1 :: "('a,'b,'c) fsm"
  fixes M2 :: "('d,'b,'c) fsm"
  assumes "observable M1"
  and     "observable M2"
  and     "minimal M1"
  and     "minimal M2"
  and     "inputs M2 = inputs M1"
  and     "outputs M2 = outputs M1"
  and     "L M1 ∩ T = L M2 ∩ T"
  and     "A ⊆ T"
  and     "transition_cover M1 A"
  and     "[] ∈ A"
  and     "preserves_convergence M1 M2 A"
shows "L M1 = L M2"
```

Here, the first six assumptions describe the usual structural requirements on reference model M_1 and system under test M_2 – not requiring any upper bound on the size of the latter – while the remaining assumptions require test suite TS to contain an initialised (i.e. containing ϵ) convergence-preserving transition cover A of M_1.

This lemma may be proven by establishing the existence of an isomorphism between M_1 and M_2, as shown in [103, Theorem 1]. The proof I provide in `Convergence.thy` roughly follows this approach, without making the isomorphism explicit. In doing so, I first show that from the properties on A it may be

Instead, the formalisation further unfolds the required properties on TS and TS'. That is, the assumption that passing TS implies passing TS' is combined with the assumption that M_2 passes TS into the assumption that M_2 passes TS', where set TS' is in turn unfolded into the relevant sets for each γ.

derived that any pair of traces α, β that converges in M_1 also converges in M_2, using structural induction on β. This immediately implies $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$. Finally, I show the converse via contradiction, using the fact that any minimal distinguishing trace $\gamma.(x/y)$ of M_1 and M_2 requires $\gamma \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ and hence implies the existence of some $\omega.(x/y) \in A$ where ω converges with γ in both FSMs. No such $\omega.(x/y) \in A$ may exist, however, as A is contained in TS, which is passed by M_2 with respect to M_1.

9.2 Required Invariants on Convergence Graphs

The remainder of this chapter describes my formalisation of convergence graphs. As introduced in Section 3.3, these constitute data structures that facilitate storage and retrieval of traces that have been established as convergent during testing. In Chapter 6, I have sketched two implementations of convergence graphs, called Empty and Standard. The H and SPY-Frameworks employ these implementations only via a small set of functions supporting the following operations:

- Lookup of traces convergent with some trace α. That is, collecting all $\beta \in [\alpha]^{M_1} \cap [\alpha]^{M_2}$ or a subset thereof, where M_1 and M_2 are the reference model and system under test, respectively.
- Insertion of a trace into the graph.
- Initialisation of a convergence graph for a given test suite.
- Merging of the classes of two traces α, β. By Lemma 2.5.2, this might include merging of all common extensions of these traces, that is, of $\alpha.\gamma, \beta.\gamma$ for some or all traces γ.

That is, the H and SPY-Frameworks view convergence graphs as an abstract data type defined by these four operations. This allows for easy substitution of implementations of convergence graphs, for example between multiple implementations of Standard.

In order to prove correctness of steps in the frameworks that employ convergence graphs via the provided operations, certain invariants over these operations are required. These invariants are defined in `Convergence_Graph.thy`. In the following, I use names `lookup`, `insert`, `initial`, and `merge` for the four operations described above.

The most fundamental invariant is the requirement that a lookup of trace α in some graph G does not return any trace that does not converge with α in M_1 or M_2. Here I additionally constrain α to be contained in both $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$, as within the H and SPY-Frameworks, lookups are only applied to traces in $\mathcal{L}(M_1)$ that are also contained in a test suite passed by M_2. This allows requiring that the lookup on such α always returns α, as it converges with itself. Thus, implementations using such `lookup` may assume that at least

one trace is returned for valid inputs. The invariant on `lookup` then is defined as follows:

$$inv^1_{\texttt{lookup}}(G) := \forall \alpha \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2).\ \alpha \in \texttt{(lookup G } \alpha\texttt{)} \wedge \forall \beta \in \texttt{(lookup G } \alpha\texttt{)}.\ \beta \in [\alpha]^{M1} \cap [\alpha]^{M2}$$

In the formalisation, this invariant is expressed in `Convergence_Graph.thy` via `(convergence_graph_lookup_invar M1 M2 lookup G)`. Note that this invariant does not require `(lookup` G α`)` to return all convergent traces, or any trace other than α at all.

The invariants for the remaining operations, which all return convergence graphs, then require that the returned convergence graph still satisfies the invariant for `lookup`. Thus, the invariant for `insert` requires that insertion of some trace α contained in both $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ into a graph G which satisfies the invariant for `lookup` retains that invariant:

$$inv^2_{\texttt{insert,lookup}} := \forall\ \texttt{G}, \alpha.\ \alpha \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \wedge inv^1_{\texttt{lookup}}(\texttt{G}) \longrightarrow inv^1_{\texttt{lookup}}(\texttt{insert G } \alpha)$$

This invariant is defined as predicate `convergence_graph_insert_invar` to be applied to M_1, M_2, and functions `lookup` and `insert`. Note that this invariant does not require the graph produced by the insertion of α into graph G to differ from G in any way.

Similarly, the invariant on `initial` only requires the returned graph to satisfy the invariant for `lookup`, assuming that the provided test suite is finite and passed by M_2 with respect to M_1. In many implementations, `insert` is merely a shorthand for repeated `insert` operations of suitable traces in the test suite into an initially empty graph.

$$inv^3_{\texttt{initial,lookup}} := \forall\ TS.\mathcal{L}(M_1) \cap TS = \mathcal{L}(M_2) \cap TS \wedge TS \text{ is finite} \longrightarrow inv^1_{\texttt{lookup}}(\texttt{initial } M_1\ TS)$$

The invariant is formalised as expression `(convergence_graph_initial_invar M1 M2 lookup initial)`. Note that in the formalisation the test suite is of type `prefix_tree` (see Appendix C).

Finally, the invariant on `merge` also merely requires preservation of the invariant on `lookup`, assuming that the graph to which `merge` is applied already satisfies the invariant and that the provided traces are safe to merge (that is, that they do converge in both M_1 and M_2).

$$inv^4_{\texttt{merge,lookup}} := \forall\ G, \alpha, \beta.\ \beta \in [\alpha]^M_1 \cap [\alpha]^M_2 \wedge inv^1_{\texttt{lookup}}(G) \longrightarrow inv^1_{\texttt{lookup}}(\texttt{merge G } \alpha\ \beta)$$

The subsequent sections show how these invariants are satisfied by implementations of Empty and Standard.

9.3 Empty Convergence Graph

As introduced in Subsection 6.3.4, EMPTY is a convergence graph that returns just $\{\alpha\}$ for lookups on any α. This requires no storage of information in the graph itself, as α is given in the query. Hence I implement EMPTY in `Empty_Convergence_Graph.thy` by type `unit`, which is inhabited by only the single value `()`.

```
type_synonym empty_cg = unit
```

As the lookup operation on EMPTY, `empty_cg_lookup`, always returns only the input trace, invariant $inv^1_{\texttt{empty\_cg\_lookup}}$ trivially holds on the only possible graph `()`. The remaining operations `empty_cg_insert`, `empty_cg_initial`, and `empty_cg_merge` can only return this `()`, and hence they trivially also satisfy their respective invariants.

9.4 Simple Convergence Graph

In `Simple_Convergence_Graph.thy`, I provide a very simple implementation of STANDARD, that is, of a convergence graph that retains the traces inserted into it and keeps track of their classes of convergent traces, supporting proper merging.

More precisely, this implementations provides a type synonym `simple_cg` for list of classes of convergent traces, where classes are in turn represented as finite sets (`fset`) of `list` values.

```
type_synonym 'a simple_cg = "'a list fset list"
```

That is, a convergence graph of type `simple_cg` is a list $[C_1, \ldots, C_k]$ where each class C_i for $0 \leq i \leq k$ is a finite set of pairwise convergent traces. Note that the type `simple_cg` is not constrained to contain only such lists that contain pairwise disjoint set. I have chosen to omit such constraints here in order to keep the implementation simple and allow experimentation with different operations on this type, not all of which ensure disjointness of contained sets.

The following subsections discuss the implementations of the lookup, insertion and merge operations of this implementation.

9.4.1 Lookup Operations

In the absence of distinctness constraints on the classes in a `simple_cg`, some trace α may be contained in several classes, and hence the simplest lookup of some trace α is performed by combining all classes containing α, inserting α itself in case no such class is found. This is implemented as `simple_cg_lookup`:

$$\texttt{simple\_cg\_lookup G}\ \alpha := \{\alpha\} \cup \left(\bigcup_{C \in G, \alpha \in C} C \right)$$

While simple to implement, this approach to lookups for α in graph G can only find β that are explicitly stored in the same classes as α. Consider graph G_1 defined as follows:

$$G_1 := [\{\alpha_1, \ldots, \alpha_k\}, \{\alpha_1.\beta_1, \ldots, \alpha_1.\beta_m\}, \{\alpha_2.\beta_1\}]$$

In this graph, `(simple_cg_lookup` G_1 $\alpha_2.\beta_1$`)` returns only the set $\{\alpha_2.\beta_1\}$. By Lemma 2.5.2, however, the fact that α_1 and α_2 converge, as per the first set in G_1, may be used to derive that $\alpha_1.\beta_1$ and $\alpha_2.\beta_1$ converge. By analogous argument, it follows that $\alpha_2.\beta_1$ converges with all traces $\alpha_i.\beta_j$ such that $1 \leq i \leq k$ and $1 \leq j \leq m$ hold. This includes set $\{\alpha_1.\beta_1, \ldots, \alpha_1.\beta_m\}$ in G_1. That is, `simple_cg_lookup` does not make use of convergences of prefixes of the looked for trace, and hence may omit returning some convergent traces in the graph.

One possible approach in strengthening the lookup via Lemma 2.5.2 would be to extend the graph until it explicitly contains all convergent traces derivable by that lemma. This would, however, often lead to an exponential increase in the size of the graph. For example, G_1 contains $k + m + 1$ traces, whereas the extension of G_1 would contain at least $k + k \cdot m$ traces, as it would include sets $\{\alpha_1, \ldots, \alpha_k\}$ and $\{\alpha_i.\beta_j \mid 1 \leq i \leq k \wedge 1 \leq j \leq m\}$.

Instead of explicitly extending the graph, I provide an alternative lookup function `simple_cg_lookup_with_conv` that applies Lemma 2.5.2 by considering for a given trace α all partitions into prefix and suffix $\alpha'.\alpha'' = \alpha$ and searching via `simple_cg` for all traces β convergent with α' in the graph. For such β, trace $\beta.\alpha''$ is included in the result, as it converges with α.

$$\texttt{simple\_cg\_lookup\_with\_conv G } \alpha := \\ \{\alpha\} \cup \{\beta.\alpha'' \mid \exists C \in G.\, \exists \alpha'.\, \alpha = \alpha'.\alpha'' \wedge \alpha', \beta \in C\}$$

Both presented lookup functions satisfy the invariant on lookups (inv^1) on some G if traces contained in the same class in G converge in both M_1 and M_2. The converse also holds, as only traces in $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ may thus converge[6].

9.4.2 Insertion Operations

Insertion of trace α into graph $G = [C_1, \ldots, C_k]$ is supported via function `simple_cg_insert'`, which returns G if it already contains a class containing α, and otherwise returns $[\{\alpha\}, C_1, \ldots, C_k]$. That is, otherwise it returns the graph obtained by adding a new class containing only α to G. This function trivially satisfies the invariant on insertions (inv^2) with respect to the two previously discussed lookup operations, as it either leaves input graph G unchanged, where G satisfies the lookup invariant by assumption, or else inserts a singleton class containing α, which is contained in $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ by assumption. As it is a singleton, this added class has no impact on the result obtained by either lookup function. Note that `simple_cg_insert'` does not try to insert the

[6] Furthermore, I show in theory file `Simple_Convergence_Graph.thy` with lemma `simple_cg_lookup_invar_with_conv_eq` that on observable FSMs and arbitrary G, $inv^1_{\texttt{simple\_cg\_lookup\_with\_conv}}(G)$ holds if and only if $inv^1_{\texttt{simple\_cg\_lookup}}(G)$ holds.

given trace into a class of convergent traces, if any such class already exists in the graph, in order to keep the implementation simple and short. Modifications of existing classes are to be performed via the merge operation discussed in the next subsection.

From `simple_cg_insert'` two functions are immediately derived. First, `(simple_cg_insert` G α`)` uses `simple_cg_insert'` to insert into G not only α but also all proper prefixes of α. Satisfaction of the insertion invariant for it derives directly from that of `simple_cg_insert'`.

Second, (`simple_cg_initial` M_1 TS) employs `simple_cg_insert'` to insert into an initially empty graph G_0 = `[]` all traces in $TS \cap \mathcal{L}(M_1)$. As G_0 by construction satisfies the lookup invariant for both lookup operations, I derive inv^3 for `simple_cg_initial` via induction on the number of traces in TS to add, where after base case G_0 each step performs `simple_cg_insert'` and hence retains satisfaction of the lookup invariant of the intermediate graph.

9.4.3 Merge Operations

Finally, function `simple_cg_merge` provides an implementation of the merging of two traces α, β in some `simple_cg` G. As an initial step, this function adds class $\{\alpha, \beta\}$ to G, representing the convergence of these traces. Thereafter, it performs the following two reduction steps until they are both no longer applicable:

1. Classes C_i, C_j, $i \neq j$, of G may be merged if there exist some traces π, τ, γ and a class C of G such that $\pi, \tau \in C$ while also $\pi.\gamma \in C_i$ and $\tau.\gamma \in C_j$ hold. That is, classes may be merged by direct application of property (1.) of Lemma 2.5.2.

2. Classes C_j, C_j, $i \neq j$, of G may be merged if there exists some $\pi \in C_i \cap C_j$, as convergence constitutes an equivalence relation on traces[7].

This implementation follows the outline given for the convergence graph of the SPY-Method described in [103], but employs simpler data structures than the union-find data structure (see [36]) discussed there.

As an example, consider the merging of traces α and β in graph $G := [\{\alpha, \pi.\omega\}, \{\beta, \tau\}, \{\alpha.\gamma, \beta.\gamma\}, \{\tau.\gamma, \pi\}, \{\beta.\gamma.\pi\}]$, resulting in the following evaluation, which reduces the number of classes from five to two:

`simple_cg_merge G` α β

$\rightsquigarrow [\{\boldsymbol{\alpha}, \beta\}, \{\boldsymbol{\alpha}, \pi.\omega\}, \{\beta, \tau\}, \{\alpha.\gamma, \beta.\gamma\}, \{\tau.\gamma, \pi\}, \{\beta.\gamma.\omega\}]$	(initial step)
$\rightsquigarrow [\{\alpha, \boldsymbol{\beta}, \pi.\omega\}, \{\boldsymbol{\beta}, \tau\}, \{\alpha.\gamma, \beta.\gamma\}, \{\tau.\gamma, \pi\}, \{\beta.\gamma.\omega\}]$	((2.) on $\boldsymbol{\alpha}$)
$\rightsquigarrow [\{\boldsymbol{\alpha}, \beta, \pi.\omega, \boldsymbol{\tau}\}, \{\boldsymbol{\alpha.\gamma}, \beta.\gamma\}, \{\boldsymbol{\tau.\gamma}, \pi\}, \{\beta.\gamma.\omega\}]$	((2.) on $\boldsymbol{\beta}$)
$\rightsquigarrow [\{\alpha, \beta, \boldsymbol{\pi.\omega}, \tau\}, \{\alpha.\gamma, \boldsymbol{\beta.\gamma}, \tau.\gamma, \boldsymbol{\pi}\}, \{\boldsymbol{\beta.\gamma.\omega}\}]$	((1.) on $\boldsymbol{\alpha}, \tau, \gamma$)
$\rightsquigarrow [\{\alpha, \beta, \pi.\omega, \tau, \beta.\gamma.\omega\}, \{\alpha.\gamma, \beta.\gamma, \tau.\gamma, \pi\}]$	((1.) on $\boldsymbol{\pi}, \boldsymbol{\beta.\gamma}, \boldsymbol{\omega}$)

[7] Note that this step is merely a special case of the previous step where $\pi = \tau$ and $\gamma = \epsilon$. It is included as a separate step in the algorithm, as it is easier to check and may quickly reduce the number of classes in certain graphs, simplifying application of the first step.

I establish invariant inv^4 for `simple_cg_merge` by induction on the number of steps performed until neither step is applicable. Note that this number is well defined, as every step merges two classes in the finite graph, thus reducing the length of the graph by one. For each type of step it holds that if classes in G contain only pairwise convergent traces, then so do merged classes C_i and C_j. By Lemma 2.5.2, it follows that the merged class $C_i \cup C_j$ also contains only pairwise convergent traces. As the other classes in G are not modified, the graph obtained by applying either step thus still satisfies the lookup invariant for both previously discussed lookup functions.

As its name implies, `simple_cg` is a very simple implementation of convergence graph STANDARD, employing data structures and functions that are simple and easy to reason about. More efficient data structures and function implementations are discussed for minimal complete DFSMs in [106, Chapters 8 and 10]. For the accompanying implementation see [105]. I have developed an analogous implementation in the `libfsmtest` open source library (see [4]).

Chapter 10

Frameworks

In this chapter, I introduce the formalisation of the three frameworks developed in Sections 6.3 to 6.5. As depicted in Fig. 7.1, each framework is formalised in a separate theory file. For each framework, the corresponding theory file implements the framework (Algorithm 17, 30, or 33) and establishes the corresponding completeness lemma (Lemma 6.3.1, 6.4.1, or 6.5.1). I exemplify this approach for the H-Framework in Section 10.1. For the SPY and Pair-Frameworks I provide short overviews in Sections 10.2 and 10.3. Appendices D.11 to D.13 list functions developed in the theory files discussed in this chapter.

10.1 H-Framework

Formalisation of the H-Framework in theory file `H_Framework.thy` consists of two steps. First, the abstract H-Condition (Lemma 4.1.7) is formalised and proven to be a sufficient condition for completeness in Subsection 10.1.1. Next, it is shown that under certain assumptions on its procedural parameters, the H-Framework generates complete test suites. To this end, Subsection 10.1.2 briefly discusses the formalisation of both the implementation of Algorithm 17 in Isabelle/HOL and Lemma 6.3.1. I omit describing here the formalised proof of Lemma 6.3.1, as it closely follows the proof given in Section 6.3.

10.1.1 Sufficiency of the Abstract H-Condition

I formalise satisfaction of conditions (1.) and (2.) described in Lemma 4.1.7 by some reference model M and implementation I for some V and m as the following predicate:

```
"satisfies_abstract_h_condition M I V m = (∀ q γ .
  q ∈ reachable_states M ⟶
  length γ ≤ Suc (m-size_r M) ⟶
  list.set γ ⊆ inputs M × outputs M  ⟶
  butlast γ ∈ LS M q ⟶
```

```
  (let traces = (V ‘ reachable_states M)
                ∪ {V q @ ω' | ω'. ω' ∈ list.set (prefixes γ)}
    in (L M ∩ traces = L I ∩ traces)
       ∧ preserves_divergence M I traces))"
```

Here, predefined function `butlast` applied to a non-empty list returns the largest proper prefix of that list, while applied to the empty list it returns the empty list. Thus, the four premises of the definition specify containment of γ in X_q as defined in Lemma 4.1.7. The `let` expression furthermore gives name `traces` to set $\{V(q) \mid q \in M\} \cup \{v_q\}.pref(\gamma)$. Note that this formalisation considers only reachable states q and hence constitutes a slight weakening of the required conditions. Furthermore, instead of a state cover, V is a state cover assignment as discussed in Section 8.3.

Formalisation of Lemma 4.1.7 now merely consists of listing the remaining assumptions on M, I, V and m:

```
lemma abstract_h_condition_exhaustiveness :
  assumes "observable M"
  and     "observable I"
  and     "minimal M"
  and     "size I ≤ m"
  and     "m ≥ size_r M"
  and     "inputs I = inputs M"
  and     "outputs I = outputs M"
  and     "is_state_cover_assignment M V"
  and     "satisfies_abstract_h_condition M I V m"
shows "L M = L I"
```

I prove this lemma by employing a standard proof for the classical H-Condition (see Lemma 4.1.2) as described in, for example, [82, Section 4.7]. This proof by contradiction assumes $\mathcal{L}(M) \neq \mathcal{L}(I)$ and derives the existence of a non-empty minimal trace to a failure τ extending $v_q = V(q)$ for some reachable state q of M. From the minimality of τ and the assumption that the abstract H-Condition holds, it follows that no failure is observed while applying the prefix of length $m - n + 1$ of τ after $V(q)$. Furthermore, by the abstract H-Condition, the traces obtained by applying V to Q and those obtained by appending non-empty prefixes of τ of length up to $m - n + 1$ to $V(q)$ are $\{I\}$-divergence-preserving with respect to M. By minimality of τ, they must also constitute $n + m - n + 1 = m + 1$ distinct traces and hence contain a pair of distinct traces α, β that converge in I, which, by assumption, has at most m states. Thus, as $\{\alpha, \beta\}$ is $\{I\}$-divergence-preserving, these traces must also converge in M. Finally, consider the distribution of $\{\alpha, \beta\}$ over the set of traces they originate from: both, one, or none of the traces are obtained by applying V to Q. In the first case, a contradiction is derived from $\{\alpha, \beta\}$ being $\{I\}$-divergence-preserving and $V(q), V(q')$ diverging in M for each pair of distinct reachable states q, q'. In each remaining case, a contradiction is derived by constructing a trace to a failure extending some $v \in \{V(q) \mid q \in Q\}$ that is shorter than τ, contradicting the minimality of the latter.

Fig. 10.1 visualises these main steps of the proof.

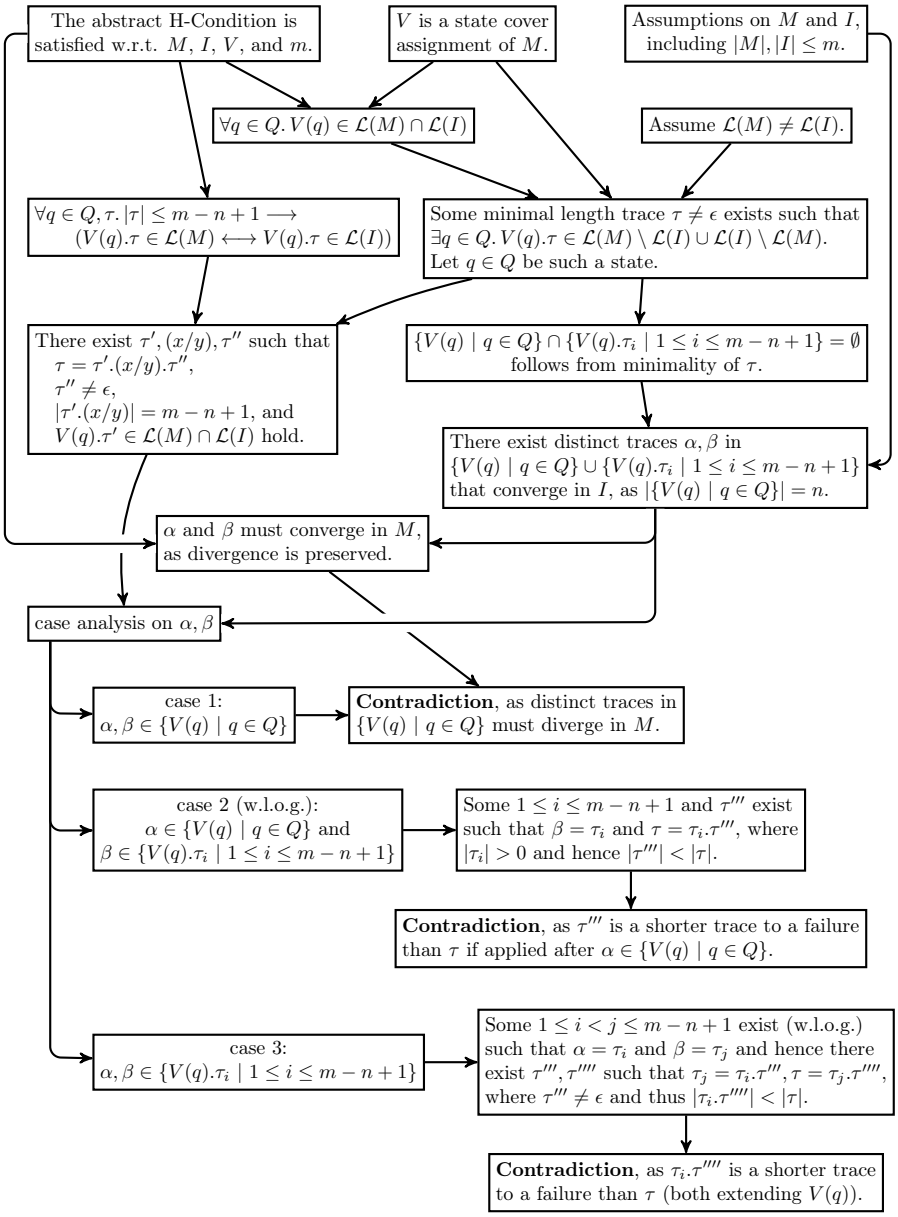


Figure 10.1: Proof sketch of lemma `abstract_h_condition_exhaustiveness` of theory file `H_Framework.thy`, which employs a proof by contradiction to establish that the abstract H-Condition is a sufficient condition for m-completeness. Assumes $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$, $|M| = n$, and – for brevity of the presentation – that all states in M are reachable. Arrows indicate the development of the argument. Trace τ_i denotes the prefix of length i of τ.

10.1.2 Completeness of the H-Framework

The implementation of the H-Framework as a higher order function follows the pseudocode implementation given in Algorithm 17. The usage of convergence graphs is made explicit by extending Algorithm 17 by four additional procedural parameters for the initialisation, insertion, merge, and lookup operations of the convergence graph to be employed (see Chapter 9). As an example of translating pseudocode functions developed in Chapter 6 to Isabelle/HOL, the implementation of Algorithm 17 is discussed in Appendix E.

The explicit handling of operations on convergence graphs also affects the formalisation of Lemma 6.3.1, as the required invariants on these operations (introduced in Section 9.2) have to be added as assumptions, which are then passed to the conditions ϕ_1^{H} to ϕ_5^{H} as required. In order to simplify the presentation, I have extracted the definitions of conditions ϕ_2^{H}, ϕ_4^{H}, and ϕ_5^{H} into predicates `separates_state_cover`, `handles_transition`, and `handles_io_pair`, respectively. In addition to the definitions of ϕ_2^{H}, ϕ_4^{H}, and ϕ_5^{H} presented in Lemma 6.3.1, these predicates also require the respective procedural parameter of the H-Framework to return a finite test suite if the test suite passed to it is finite. This property is required to establish finiteness of test suites generated by the H-Framework and has been omitted in Section 6.3 for brevity. Conditions ϕ_1^{H} and ϕ_3^{H} are sufficiently simple to be stated directly. Employing these changes, Lemma 6.3.1 is formalised as follows, abbreviating the test suite obtained by application of the `h_framework` as `?TS` (the full definition of which I omit here due to space constraints):

```
lemma h_framework_completeness_and_finiteness :
  fixes M :: "('a::linorder,'b::linorder,'c::linorder) fsm"
  fixes I :: "('e,'b,'c) fsm"
  fixes cg_insert :: "('d ⇒ ('b×'c) list ⇒ 'd)"
  assumes "observable M"  and  "observable I"  and  "minimal M"
  and     "minimal I"  and  "size_r M ≤ m"  and  "size I ≤ m"
  and     "inputs I = inputs M"  and  "outputs I = outputs M"
  and     "is_state_cover_assignment M (get_state_cover M)"
  and     "⋀X. set X = set(sort_transitions M (get_state_cover M) X)"
  and     "convergence_graph_initial_invar M I cg_lookup cg_initial"
  and     "convergence_graph_insert_invar M I cg_lookup cg_insert"
  and     "convergence_graph_merge_invar M I cg_lookup cg_merge"
  and     "separates_state_cover
             handle_state_cover M I cg_initial cg_insert cg_lookup"
  and     "handles_transition handle_unverified_transition M I
             (get_state_cover M)
             (fst (handle_state_cover M (get_state_cover M)
                                        cg_initial cg_insert cg_lookup))
             cg_insert cg_lookup cg_merge"
  and     "handles_io_pair
             handle_unverified_io_pair M I cg_insert cg_lookup"
shows "(L M = L I) ⟷ ((L M ∩ set ?TS) = (L I ∩ set ?TS))"
and   "finite_tree ?TS"
```

The mechanised proof of this lemma formalises the natural language proof given for Lemma 6.3.1 in Section 6.3. Finiteness of the generated test suite follows from an analogous argument using the updated assumptions, as the test suite initially obtained via HANDLESTATECOVER is finite by ϕ_2^{H} and then extended by finitely many applications of HANDLEUNVERIFIEDTRANSITION and HANDLEUNDEFINEDIOPAIR, which retain finiteness by ϕ_4^{H} and ϕ_5^{H}, respectively.

10.2 SPY-Framework

As discussed in Section 6.4, the SPY-Framework is very similar to the H-Framework and differs from the latter only in always merging $\upsilon_q.(x/y)$ and $\upsilon_{q'}$ after verifying a transition (q, x, y, q'), and in deriving completeness via the SPY-Condition (Corollary 4.2.2). Thus, many aspects of the formalisation of the H-Framework may be reused in formalising the SPY-Framework, and the implementation of the SPY-Framework as function `spy_framework` in theory file `SPY_Framework.thy` closely resembles that of function `h_framework` described in Appendix E.

The sufficient condition for completeness used in Lemma 6.4.1 has already been formalised in Subsection 9.1.3.

10.3 Pair-Framework

Introduced in Section 6.5, the Pair-Framework is a simpler framework that does not directly support exploitation of convergence and is suitable for implementing the W, HSI, and H-Methods. It is implemented as function `pair_framework` in `Pair_Framework.thy`, formalising Algorithm 33. Completeness is established by formalising Lemma 6.5.1 as lemma `pair_framework_completeness`, which shows that under certain assumptions on its procedural parameters, `pair_framework` generates a test suite satisfying the weakened H-Condition introduced as Lemma 4.1.3. In order to allow future reuse, the latter lemma is also formalised in `Pair_Framework.thy` as lemma `h_condition_completeness`, established by deriving the weakened H-Condition from the abstract H-Condition (Lemma 4.1.7). Finiteness of test suites generated by `pair_framework` is established in lemma `pair_framework_finiteness`.

Chapter 11

Test Strategies

The final steps of formalising the definitions and results of Part II consist in first providing concrete implementations of the functions discussed in Chapter 6 to be passed as arguments for the procedural parameters of the H, SPY, and Pair-Frameworks, followed by implementing the test strategies selected in Section 5.1 via the frameworks, and finally establishing the completeness and finiteness properties of test suites generated by such implementations.

11.1 Intermediate Implementations

The first of these steps is straightforward, as it essentially consists only of translating most of the algorithms introduced in Chapter 6 into Isabelle/HOL functions. Collected in theory file `Intermediate_Implementations.thy`, these translations differ from the pseudocode algorithms of Chapter 6 only in a more explicit handling of the initialisation, insertion, lookup, and merge operations on convergence graphs, and in the introduction of some further heuristic functions omitted in Chapter 6 for brevity.

For example, two additional heuristics govern the selection of traces in DISTRIBUTEEXTENSION(M, α, γ, TS, G) (Algorithm 1), as the algorithm given in Section 3.3 does not provide instructions on how a trace β is selected from CG-LOOKUP(G,α) in order to be appended with γ. Both heuristics try to find some β such that inserting $\beta.\gamma$ into test suite (tree) TS adds as few new edges as possible, avoiding the addition of new branches (that is, increasing the number of distinct maximal traces) if at all possible. The heuristics differ in how they count edges and branches of TS: `append_heuristic_io` views TS as it is implemented, that is, as a tree whose edges are labelled with IO-pairs, whereas `append_heuristic_input` considers only the input projection of TS. Thus, in the former, adding some trace 00/11 to {00/10} would introduce a new branch, whereas in the latter it would not, since both traces share the same input portion. Depending on usage, `append_heuristic_input` may thus lead to more branches in the test suite, but fewer distinct input sequences.

Appendix D.14 provides an overview on the functions implemented in theory file `Intermediate_Implementations.thy` and references the algorithms of Chapter 6 they implement. Note that this theory files does not contain some algorithms of Chapter 6 that either have been implemented in previously discussed theory files, such as the frameworks themselves, or that are specific to a single test strategy and not reused in implementing other strategies, such as S-Heuristic (Algorithm 28), which is employed only in implementing the $S_{partial}$-Method. The latter type of functions are listed in Appendices D.16 to D.18.

11.2 Intermediate Frameworks

As described in Chapter 6 and visualised in Fig. 6.2, Fig. 6.3, and Fig. 6.4, there exists significant overlap between the implementations of certain test strategies. For example, in the H-Framework, the W, Wp, and HSI-Methods only differ in the heuristic used in getDistSetForLength. For all other procedural parameters of the H-Framework, they employ the same functions. Similarly, in the same framework the implementations of the H, SPYH, and $S_{partial}$-Methods differ only in the heuristic they employ to decide whether to establish convergence for some trace to handle in handleUT-Dynamic[1].

Thus, I have implemented in theory file `Intermediate_Frameworks.thy` a selection of *intermediate* frameworks that partially apply the H, SPY, and Pair-Frameworks in order to simplify implementations and completeness proofs on groups of test strategy implementations. For example, for the H-Framework, the following intermediate frameworks are sufficient to allow implementation of all required strategies. These all apply getStateCoverByBFS (Algorithm 18) and handleUndefinedIOPair (Algorithm 29) to the H-Framework. Appendix D.15 lists all available intermediate frameworks.

- `h_framework_dynamic`

  applies handleStateCover-Dynamic (Algorithm 19) and handleUT-Dynamic (Algorithm 27). Unverified transitions are sorted via SPYH-sortTransitions (Algorithm 24)[2]. Operations on convergence graphs are implemented using convergence graph Standard (implemented as `simple_cg`, see Section 9.4). Procedural parameter doEstablishConvergence of handleUT-Dynamic is added as the sole remaining procedural parameter of the framework. Thus, this framework supports implementing the H, SPYH, and $S_{partial}$-Methods.

[1] In Fig. 6.3 the H-Method also differs from the SPYH and $S_{partial}$-Methods by employing convergence graph Empty instead of Standard. However, as the H-Method never merges any convergent classes in its implementation using the H-Framework, it may equivalently be implemented using Standard as implemented in Section 9.4. I have chosen this latter implementation here, as it allows grouping the H-Method with the SPYH and $S_{partial}$-Methods in the H-Framework.

[2] The H-Method does not prescribe a specific sorting on the unverified traces and hence may also use SPYH-sortTransitions. The $S_{partial}$-Method coincides with the SPYH-Method in the sorting algorithm by definition (see Subsection 6.3.4).

- `h_framework_static_with_empty_graph`

  applies HANDLESTATECOVER-STATIC (Algorithm 20) and HANDLEUT-STATIC (Algorithm 25) and does not sort unverified transition. Operations on convergence graphs are implemented using convergence graph EMPTY (that is, the operations introduced in Section 9.3). Procedural parameter GETDISTSETFORLENGTH of HANDLEUT-STATIC is added as the sole remaining procedural parameter of the framework. Thus, this framework supports implementing the W, Wp, and HSI-Methods.

- `h_framework_static_with_simple_graph`

  is identical to the previous intermediate framework except in that it employs operations on convergence graph STANDARD as introduced in Section 9.4. Thus, this framework supports implementing the SPY-Method for arbitrary functions for computing harmonised state identifiers.

Fig. 11.1 and 11.2 visualise how establishing completeness properties on these intermediate frameworks simplifies establishing completeness of concrete test strategies implemented with their help. For example, the fourth block from the bottom left hand side of Fig. 11.1 represents the result that test suites obtained by `h_framework_dynamic` applied to some minimal OFSM M, upper bound m and concrete implementation d of DOESTABLISHCONVERGENCE are m-complete. This is established in `Intermediate_Frameworks.thy` as lemma `h_framework_dynamic_completeness_and_finiteness`.

Fig. 11.1 shows the main steps how this result is established in the formalisation. The proof relies on the sufficiency condition of the H-Framework (Lemma 6.3.1, see Subsection 10.1.2) and derives completeness by showing how the functions supplied to the procedural parameters of the H-Framework by `h_framework_dynamic` (abbreviated as `f1` to `f5` in Fig. 11.1) satisfy conditions ϕ_1^{H} to ϕ_5^{H}. Of these, ϕ_1^{H} requires showing that GETSTATECOVERBYBFS (Algorithm 18) computes a state cover, which I have already established in Section 8.3 for its formalisation as function `get_state_cover_assignment`. Next, ϕ_3^{H} requires SPYH-SORTTRANSITIONS (Algorithm 24) to at most change the order of the given collection of transitions, without adding or removing transitions. This depends solely on the employed underlying sorting algorithm and is not affected by the weights assigned to transitions. In my formalisation, I employ merge-sort function `mergesort_by_rel` of AFP-entry [63], which provides the required property. Condition ϕ_2^{H} thereafter requires function HANDLESTATECOVER-DYNAMIC (Algorithm 19) to separate the traces in the state cover and initialise a valid convergence graph. This is established in lemma `handle_state_cover_dynamic_separates_state_cover` of theory file `Intermediate_Implementations.thy` analogous to the description in Subsection 6.3.2, relying on properties of `SPYH-distinguish`. Similarly, establishing satisfaction of condition ϕ_5^{H} for HANDLEUNDEFINEDIOPAIR (Algorithm 29) follows the description given in Subsection 6.3.5 and is formalised in lemma `verifies_io_pair_handled`.

Finally, establishing condition ϕ_4^{H} on HANDLEUT-DYNAMIC requires a more elaborate argument, as sketched in Subsection 6.3.4. This follows from the nesting of several algorithms: HANDLEUT-DYNAMIC calls `distinguish_from_set` (Algorithm 12), which in turn employs SPYH-DISTINGUISH (Algorithm 9), which uses BESTPREFIXOFSEPSEQ (Algorithm 13), which in the formalisation exhibits an additional procedural parameter `getDist` for a function computing distinguishing traces. Function `h_framework_dynamic` here employs the function based on OFSM-Tables introduced in Section 8.6, abbreviated as Δ_{OFSM} in Fig. 11.1. Many of these functions also employ DISTRIBUTEEXTENSION (Algorithm 1) in order to insert traces into the test suite.

Recall that ϕ_4^{H} also depends on the function employed to handle state covers, which in the case of `h_framework_dynamic` is HANDLESTATECOVER-DYNAMIC. Lemma `handleUT_dynamic_handles_transition` provides an interface lemma abstracting from the use of a concrete implementation for `getDist` and the handling of state covers, and instead shows that ϕ_4^{H} holds for any such implementations as long as the former generates valid distinguishing traces and the test suite passed to `h_framework_dynamic` ensures preservation of divergence in the state cover assignment used. Fig. 11.1 visualises these assumptions via dotted lines and shows how `handleUT_dynamic_handles_transition` is derived from the properties of its constituent functions. First, it is established that the test suite returned by HANDLESTATECOVER-DYNAMIC is finite and contains the test suite supplied as input, by showing that this holds initially for DISTRIBUTEEXTENSION and in turn for SPYH-DISTINGUISH, DISTINGUISHFROMSET, and finally HANDLESTATECOVER-DYNAMIC. Next, from the assumption on `getDist` (denoted Δ in Fig. 11.1), it follows that calls to BESTPREFIXOFSEPSEQ do return prefixes of distinguishing traces. Furthermore, DISTRIBUTEEXTENSION$(M, \alpha, \gamma, TS, G)$ by definition inserts into TS some $\beta.\gamma$ such that β converges with α in the reference model and all $I \in \mathcal{F}(M, m)_{TS}$ if G is a valid convergence graph. These properties suffice to derive the fact that calls to SPYH-DISTINGUISH(α, X, TS, G) augment TS to separate α from all diverging traces in X, as for each not already separated trace $\upsilon \in X$ a new separating trace is selected via `getDist` and appended to convergent traces of α and υ, respectively, via DISTRIBUTEEXTENSION. From this and an induction on k it follows that DISTINGUISHFROMSET$(\alpha, \beta, V, X, TS, G, k)$ establishes preservation of divergence on $X \cup \{\alpha, \beta\}.\mathit{pref}(\gamma)$ for all traces γ of length at most k whose proper prefix is in $\mathcal{L}_M(M\text{-after-}\alpha)$. Hence, from the application of DISTINGUISHFROMSET in HANDLEUT-DYNAMIC it can be derived that ϕ_4^{H} holds for HANDLEUT-DYNAMIC regardless of the value returned by the heuristic DOESTABLISHCONVERGENCE. Preservation of the validity of supplied convergence graphs follows from a similar argument through the various constituent functions of HANDLEUT-DYNAMIC. Thus, finally, lemma `handleUT_dynamic_handles_transition` follows and from this it follows that condition ϕ_4^{H} holds for the implementation of `h_framework_dynamic`, since the assumptions on the former lemma have been established for Δ_{OFSM} in Section 8.6 and for the preservation of divergence via ϕ_3^{H}.

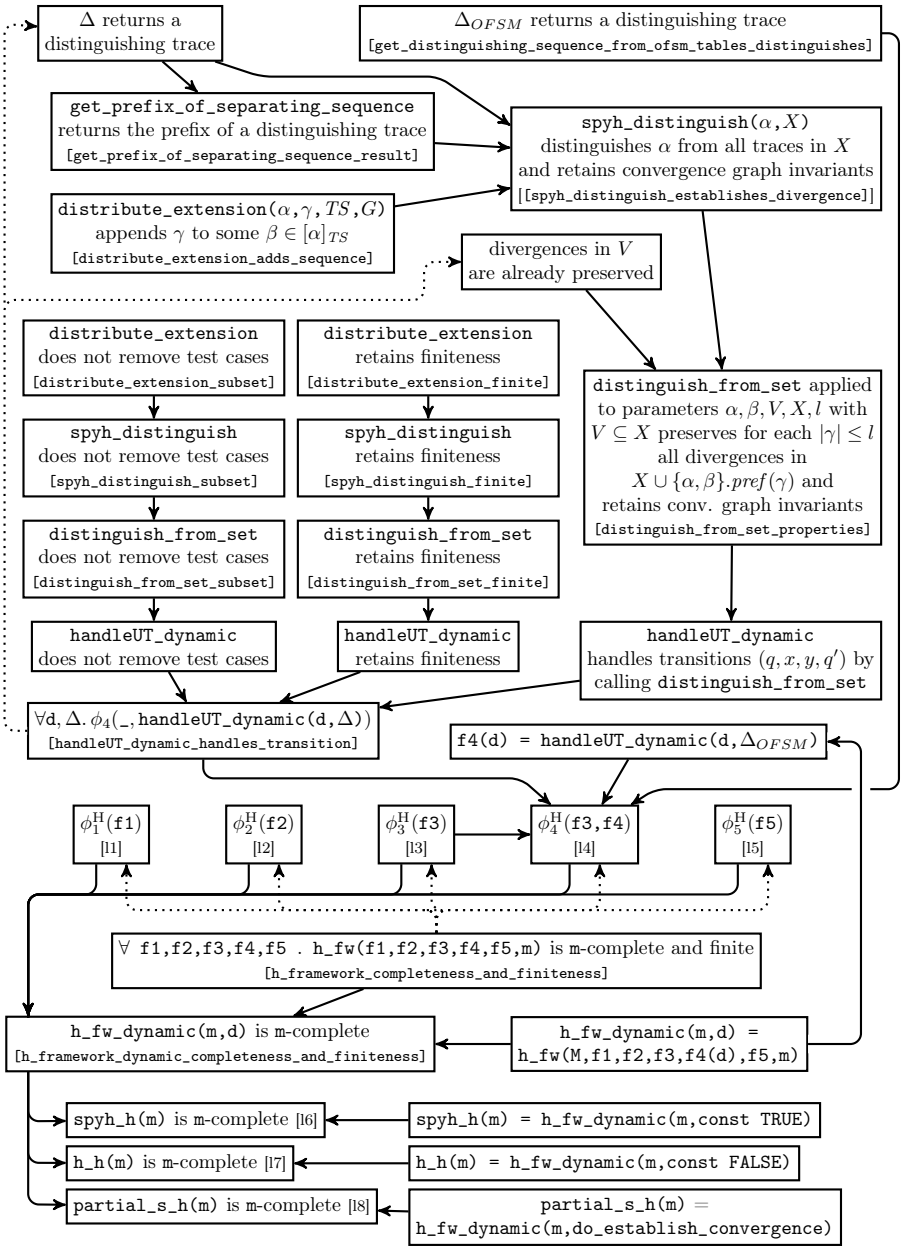


Figure 11.1: Schematic overview of primary lemmata used in establishing completeness of the H, SPYH, and $S_{partial}$-Methods implemented via the H-Framework. Solid arrows represent implications, dotted arrows indicate assumptions. Names in brackets indicate lemmata of the formalisation. Abbreviations `[l1]` to `[l8]` are detailed in Table 11.1. Prefix `h_fw` abbreviates `h_framework`. Δ_{OFSM} is developed in Section 8.6. Only selected parameters are shown.

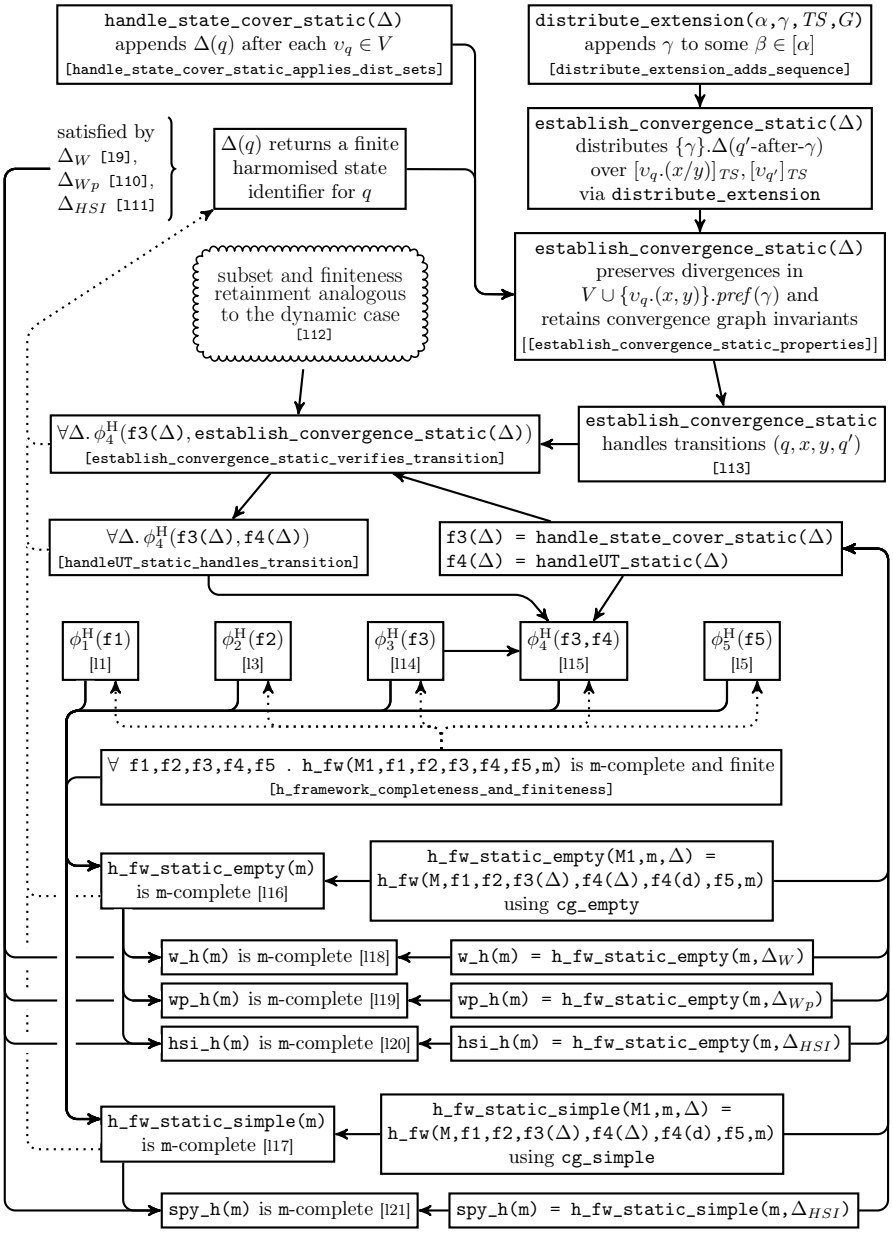


Figure 11.2: Schematic overview of the most important lemmata employed in establishing completeness of test suites generated by the W, Wp, HSI, and SPY-Methods implemented using the H-Framework. Notation is analogous to Fig. 11.1. Functions Δ_W, Δ_{Wp}, and Δ_{HSI} are defined in Algorithms 23, 26, and 22, respectively.

Table 11.1: Full names of lemmata employed in Fig. 11.1 and Fig. 11.2.

abbr.	name of lemmata in formalisation
[l1]	`get_state_cover_assignment_is_state_cover_assignment`
[l2]	`handle_state_cover_dynamic_separates_state_cover`
[l3]	`sort_unverified_transitions_by_state_cover_length_retains_set`
[l4]	`handleUT_dynamic_handles_transition`
[l5]	`verifies_io_pair_handled`
[l6]	`spyh_method_via_spy_framework_completeness_and_finiteness`
[l7]	`h_method_via_h_framework_completeness_and_finiteness`
[l8]	`partial_s_method_via_h_framework_completeness_and_finiteness`
[l9]	`distinguishing_set_distinguishes`, `distinguishing_set_finite`
[l10]	`distinguishing_set_or_state_identifier_distinguishes`, `distinguishing_set_or_state_identifier_finite`
[l11]	`get_HSI_distinguishes`, `get_HSI_finite`
[l12]	`establish_convergence_static_subset`, `establish_convergence_static_finite`
[l13]	`establish_convergence_static_verifies_transition`
[l14]	`handle_state_cover_static_separates_state_cover`
[l15]	`handleUT_static_handles_transition`
[l16]	`h_framework_static_with_empty_graph_completeness_and_finiteness`
[l17]	`h_framework_static_with_simple_graph_completeness_and_finiteness`
[l18]	`w_method_via_h_framework_completeness_and_finiteness`
[l19]	`wp_method_via_h_framework_completeness_and_finiteness`
[l20]	`hsi_method_via_h_framework_completeness_and_finiteness`
[l21]	`spy_method_via_h_framework_completeness_and_finiteness`

11.3 Test Strategy Implementations

For each of the test strategies discussed in Section 5.1, a theory file with suffix `_Implementations.thy` contains the implementations of the respective strategy in using the intermediate frameworks (see Fig 7.1). These implementations supply concrete implementations to the remaining procedural parameters as indicated in Fig. 6.2, Fig. 6.3, and Fig. 6.4. Each implementation exhibits at least two inputs: reference model M and upper bound m. Some implementations exhibit further Boolean parameters that govern choices of heuristics (see Section 11.1) and whether the insertion of traces into the test suite should trigger the additional insertion of certain related traces[3].

Completeness and finiteness proofs on these implementations reduce to employing the completeness and finiteness results of the used intermediate frameworks and showing that any remaining procedural parameters satisfy the assumptions stated in these results. For example, Fig. 11.1 shows how completeness of the implementations of the H, SPYH, and $S_{partial}$-Methods using the H-Framework can be directly derived from the completeness property of intermediate framework `h_framework_dynamic`, which imposes no requirements on on the single remaining procedural parameter doEstablishConvergence. Similarly, Fig. 11.2 shows that in order to prove complete the test suites generated by implementations of the W, Wp, HSI, and SPY-Methods using the H-Framework using intermediate frameworks `h_framework_static_with_empty_graph` and `h_framework_static_with_simple_graph`, it suffices to show that the functions they supply to procedural parameter getDistSetForLength return (finite) harmonised state identifiers.

This approach furthermore minimises the effort required to implement variations of test strategies that differ from the already implemented strategies only in implementations of one or more procedural parameters of intermediate frameworks. For example, an alternative implementation of the W-Method might use a more sophisticated method to compute characterisation sets than getCharSet (Algorithm 23) such as getCharSet-reduced, implemented in Algorithm 41. In contrast to getCharSet, this function avoids inserting distinguishing trace getShortestDistTrace(M, q, q') into the intermediate characterisation set if the latter already contains a distinguishing trace for q and q', thus avoiding superfluous entries. An implementation of the W-Method using getCharSet-reduced is obtained by simply passing getCharSet-reduced instead of getCharSet to `h_framework_static_with_empty_graph`. The establishment of completeness and correctness of this new strategy furthermore reduces to proving that getCharSet-reduced returns finite harmonised state identifiers.

Similarly, the implementation of the H-Method using the Pair-Framework as introduced in Section 6.5 can be improved by adding a heuristic to get-

[3]For example, insertion of $\bar{x}/\bar{y}$ might trigger the insertion of all possible traces with the same input portion $\bar{x}$. This completion may improve the effect of some heuristics and does not increase the size of the test suite if test cases are to be eventually converted to input sequences.

Algorithm 41: getCharSet-reduced(M, q)

```
1 W ← ∅
2 foreach (q, q') ∈ Q × Q do
3     if there exists no γ ∈ pref(W) ∩ Δ_M(q, q') then
4         W ← W ∪ {getShortestDistTrace(M, q, q')}
5 return W
```

DistTraceIfReq (Algorithm 38) that tries to find a distinguishing trace requiring as few new edges and branches in the test suite (tree) as possible. In `H_Method_Implementations.thy`, I have implemented this alternative function as `add_cheapest_distinguishing_trace`. It essentially follows the heuristic already implemented for the H and SPY-Frameworks in bestPrefixOfSepSeq (Algorithm 13), without the need to consider convergence graphs, as the Pair-Framework does not employ these.

The following implementations of test strategies are available in my formalisation, where curly brackets indicate a group of distinct prefixes for a given suffix:

- `{w,wp,hsi,h,spy,spyh,partial_s}_method_via_h_framework`

  Implementations of test strategies as described for the H-Framework in Section 10.1.

- `w_method_via_h_framework_2`

  Implementation of the W-Method in the H-Framework using function getCharSet-reduced as described above.

- `{w,wp,hsi,spy,spyh}_method_via_spy_framework`

  Implementations of test strategies as described for the SPY-Framework in Section 10.2.

- `{w,hsi,h}_method_via_pair_framework`

  Implementations of test strategies as described for the Pair-Framework in Section 10.3.

- `h_method_via_pair_framework_2`

  Implementation of the H-Method in the Pair-Framework with optional completions of inserted traces.

- `h_method_via_pair_framework_3`

  Implementation of the H-Method in the Pair-Framework with further optional completions of inserted traces and employing heuristic function `add_cheapest_distinguishing_trace` described above.

Part IV

Verified Executable Implementations

Chapter 12

Code Generation

This chapter discusses the generation of trustworthy executable implementations of the strategies discussed in Section 11.3 via the code generation capabilities of Isabelle/HOL. First, Section 12.1 introduces these capabilities and provides examples from my formalisation. Next, Sections 12.2 to 12.4 show examples of refinements performed to obtain more efficient implementations that retain correctness properties. Finally, Section 12.5 discusses the generation of implementations to be employed by the tools introduced in Chapter 13.

12.1 Code Generation With Isabelle/HOL

The Isabelle/HOL distribution contains code generation facilities able to create executable implementations from many definitions, targeting several functional programming languages including Haskell, Standard ML (SML), OCaml and Scala. A partial correctness proof for this translation is given in [42], showing that if the generated program terminates in the target language without an error or exception being thrown, then it returns the same value as the implementation in Isabelle/HOL[1]. In [41, 42] it is also demonstrated how the code generator may be employed to facilitate program and data refinement in order to obtain more efficient implementation, which is applied to my formalisation in subsequent sections. An extended tutorial on the code generator is given in [39].

The code generation capabilities have been employed to obtain verified implementations of a wide variety of algorithms, including LTL model-checkers (see [30]), theorem provers (see [98]), SAT solvers and SAT certificate checkers (see [34, 65]), linear mixed integer arithmetic solvers (see [15]), computations of enclosures of solutions of ordinary differential equations (see [54]), factorisation algorithms (see [25]), or file comparison algorithms (see [118]). In many cases the obtained implementations are sufficiently efficient to be applied to practical

[1] Note that this proof has been established on paper. Mechanised verification of the code generation and subsequent compilation for Isabelle/HOL has been performed for the target language CakeML in [52].

problems. The development of executable implementations is also already of use during the development of the formalisation itself, as the Quickcheck tool integrated in Isabelle/HOL (see [21]) uses these to try to find counterexamples for stated but not-yet-proven lemmata. Quickly finding such counterexamples helps identifying invalid conjectures and thus avoids lengthy unsuccessful attempts at proving them.

Code generation is based on the equations used in Isabelle/HOL definitions. For example, recall the formalisation of predicate `observable` from Subsection 8.2.4, which checks whether a given FSM is observable:

```
fun observable :: "('a,'b,'c) fsm ⇒ bool" where
  "observable M =
    (∀ t1 ∈ transitions M . ∀ t2 ∈ transitions M .
      (t_source t1 = t_source t2 ∧ t_input t1 = t_input t2 ∧
       t_output t1 = t_output t2)
        ⟶ t_target t1 = t_target t2)"
```

This is translated into the following function in Haskell:

```
observable ::
  forall a b c. (Eq a, Eq b, Eq c) => Fsm a b c -> Bool;
observable m =
  ball (transitions m)
    (\ t1 ->
      ball (transitions m)
        (\ t2 ->
          (if fst t1 == fst t2 &&
                fst (snd t1) == fst (snd t2) &&
                  fst (snd (snd t1)) == fst (snd (snd t2))
            then snd (snd (snd t1)) == snd (snd (snd t2))
            else True)));
```

That is, both quantifications over the set of transitions of the given FSM are translated into calls of function `ball`, which checks whether a given predicate holds for all elements of a set (see below). Furthermore, an implication $P \longrightarrow Q$ is translated into a statement `(if P then Q else True)`. Finally, the projections for the components of a transition, such as `t_source`, are unfolded to the operations on tuples they represent[2]. Note that the translation to Haskell also makes explicit that arbitrary types in Isabelle/HOL by default support equality checks via `=`, which in Haskell requires all three type parameters of `observable` to instantiate class `Eq`. When translating to languages that do not themselves support type classes directly, the functions defined in type class instantiations are added as procedural parameters to the generated functions (see [42]).

The translation of $(\forall x \in A.\, P(x))$ into `(ball A P)` is predefined in Isabelle/HOL and the implementation of `ball` itself depends on the representation of sets (see also Section 12.4). The default generated implementation of

[2]Recall that transitions are 4-tuples (q, x, y, q'), which in Isabelle/HOL are represented as nested pairs $(q, (x, (y, q')))$. First and second element of a pair are accessed in both Isabelle/HOL and Haskell via functions `fst` and `snd`, respectively.

the type of sets in Isabelle/HOL uses lists – which in Isabelle/HOL are of finite length – to represent either a finite set `(Set xs)` whose elements are stored in list xs, or a set `(Coset xs)` that is the complement of `(Set xs)`. This type is exported to Haskell as follows:

```
data Set a = Set [a] | Coset [a];
```

Of these, `ball` is only well defined on `(Set xs)`, in which case it employs the predefined function `all` to check satisfaction of the predicate for all elements:

```
ball :: forall a. Set a -> (a -> Bool) -> Bool;
ball (Set xs) p = all p xs;
```

On a `(Coset xs)`, `ball` is not defined, as the construction of the complement of `(Set xs)` would require generating the universe of all values of the same type as those in list xs, which the code generator by default only supports for types with a finite number of values[3].

By this constraint, the definition of function `h` described in Section 8.1 is not immediately suitable for code generation, as for `(h M (q,x))` it implicitly quantifies over all possible y and q' such that (q, x, y, q') is a transition of M.

```
fun h :: "('a,'b,'c) fsm_impl ⇒ ('a × 'b) ⇒ ('c × 'a) set" where
  "h M (q,x) = { (y,q') . (q,x,y,q') ∈ transitions M }"
```

In order to enable code generation without the need to always consider the suitability for this purpose while developing a definition, the code generator allows specification of separate *code equations*. These are alternative definitions to be employed by the code generator instead of the original definition of some function, and the code generator requires proof that both definitions are equivalent. For example, theory file `FSM_Impl.thy` defines the following lemma to provide a definition of `h` suitable for code generation:

```
lemma h_code[code] :
  "h M (q,x) = (let
     m = set_as_map (image (λ(q,x,y,q'). ((q,x),y,q')) (transitions M))
  in (case m (q,x) of Some yqs ⇒ yqs | None ⇒ {}))"
```

Here, `set_as_map` is a function (suitable for code generation) that obtains for a given set A of pairs a mapping f such that $f(x) := \{y \mid (x, y) \in A\}$. Attribute `code` of the lemma indicates its use as a code equation for the code generator[4].

12.2 Data Refinement: Updated FSM Data Type

The data type for FSMs as introduced in Section 8.1 does store transitions only in a single set. Hence, each call to `h` presented above requires renewed

[3] More precisely, the code generator in the absence of alternative instructions only supports generating the universe of some type if that type instantiates class `enum`.

[4] Note that a simpler implementation of `h` might be achieved by `filter` and `image` operations on sets. I have chosen the presented implementation to highlight the construction of an intermediate mapping from the transitions of the FSM, which is repeated for each call to `h` and hence might allow for common subexpression elimination (see [74]).

non-trivial computation to select the desired entries from this set. Furthermore, many functions developed in Part III repeatedly employ `h` or the related `h_obs` (see Section 8.1), so that each implemented test strategy performs a large number of calls to these functions. In order to avoid repeated computation of the same intermediate steps, in particular due to repeated calls with the same inputs during execution of a test strategy, I introduce in theory file `FSM_Code_Datatype.thy` a refinement of data type `fsm_impl` (and hence also `fsm`) that stores the views on the transition relation required by `h` and `h_obs` as mappings, allowing these functions to be implemented as simple lookups.

First, data type `fsm_with_precomputations_impl` exhibits a single constructor `FSMWPI` that essentially extends constructor `FSMI` of `fsm_impl` with mappings `h_wpi` and `h_obs_wpi`[5].

```
datatype ('a,'b,'c) fsm_with_precomputations_impl =
      FSMWPI (initial_wpi : 'a)
             (states_wpi : "'a set")
             (inputs_wpi : "'b set")
             (outputs_wpi : "'c set")
             (transitions_wpi : "('a × 'b × 'c × 'a) set")
             (h_wpi : "(('a × 'b), ('c × 'a) set) mapping")
             (h_obs_wpi: "('a × 'b, ('c,'a) mapping) mapping")
```

From this type, a type of well-formed FSMs `fsm_with_precomputations` is obtained analogous to the definition of type `fsm` in Section 8.2 via a well-formedness predicate. In contrast to `fsm`, this predicate here also requires `h_wpi` and `h_obs_wpi` to be well-formed, for example by requiring that if a value exists in `(h_wpi M)` for key `(q,x)`, then this is the set produced by `(h M (q,x))`, and that if no such value exists, then `(h M (q,x))` would return the empty set.

The code generator is instructed to employ `fsm_with_precomputations` as a refinement of `fsm_impl` by adding translations between the two types introduced above, providing a function `FSMWP` for obtaining an `fsm_impl` from an `fsm_with_precomputations` – which simply maps the first five fields of the latter to the parameters of constructor `FSMI` of the former – and finally configuring the code generator to replace constructor `FSMI` with this function using `code_datatype`. For more details on this approach see Section 3.2 of [39].

```
definition FSMWP ::
  "('a,'b,'c) fsm_with_precomputations ⇒ ('a,'b,'c) fsm_impl"
  where
  "FSMWP M = FSMI (initial_wp M)
                  (states_wp M)
                  (inputs_wp M)
                  (outputs_wp M)
                  (transitions_wp M)"

code_datatype FSMWP
```

[5] The mappings are not stored as Isabelle/HOL maps, which are regular functions, but as values of type `mapping`, which serves as an abstract interface for maps to be employed in code generation (see Section 12.4).

Values of type `fsm_impl` created using `FSMWP` retain well-formedness as described in Section 8.2 by the well-formedness of `fsm_with_precomputations`, which also allows replacing the previously developed implementation of `h` with a lookup as intended. The replacement of the previous code equation for `h` is indicated to the code generator via attribute `code drop`.

```
declare [[code drop: FSM_Impl.h]]
lemma[code] : "h (FSMWP M) = (λ (q,x) .
  case Mapping.lookup (h_wp M) (q,x) of Some yqs ⇒ yqs | None ⇒ {})"
```

Finally, due to the replacement of `FSMI` with `FSMWP`, functions of theory file `FSM_Impl.thy` that have directly used the former constructor have to be given code equations implementing them using the latter. In particular, it has to be ensured that the extended well-formedness property of the new data type is retained during construction of FSMs.

Note that no changes to `FSM_Impl.thy` or `FSM.thy` have been required to implement this refinement. No proof in the theory files discussed in Part III is to be modified and the results established therein automatically hold for the updated implementations.

12.3 Program Refinement: OFSM-Tables

As discussed in Subsection 8.5.1, the computation of the k-th OFSM-Table via function `ofsm_table` as implemented in `Minimisation.thy` is simple and intuitive but not efficient, as during its recursive definition it does not store intermediate results and hence repeatedly computes preceding tables. This negatively impacts the performance of both minimisation (see Subsection 8.5.2) and the computation of distinguishing traces as implemented in Section 8.6, which make heavy use of OFSM-Tables.

Theory file `OFSM_Table_Refinement.thy` refines `ofsm_table` by introducing a function `compute_ofsm_tables` that creates a list containing the first to k-th OFSM-Table by iteratively applying a function `next_ofsm_table` to obtain the $(i+1)$-th from the already computed i-th table without the need to repeatedly compute preceding tables. More precisely, `(compute_ofsm_tables M k)` effectively returns the same list as `[initial_ofsm_table, ofsm_table M (λq. states M) 1, ..., ofsm_table M (λq. states M) k]`, where helper function `initial_ofsm_table` assigns to each input the set of all states of M.

This function enables refining the minimisation algorithm presented in Subsection 8.5.2 by computing the $(|M|-1)$-th table using `compute_ofsm_tables` instead of employing `ofsm_table_fix`. Both tables must be identical, as there cannot exist more than $|M|$ distinct classes in an OFSM-Table and each step from the i-th to the $(i+1)$-th table either increases the number of distinct classes or leaves the table unchanged, in which case all subsequent tables are identical.

The computation of a distinguishing traces can be refined analogously. Note that after such refinement, the computation of OFSM-Tables is still repeated for each call to `get_distinguishing_sequence_from_ofsm_tables`. This constitutes a further opportunity for refinement by computing all tables up to

the $(|M| - 1)$-th table once and reusing this list of tables in subsequent calls. For this purpose, `OFSM_Table_Refinement.thy` furthermore defines a version of `get_distinguishing_sequence_from_ofsm_tables` that uses a given list of OFSM-Tables instead of computing them itself. Via refinement, all calls to `get_distinguishing_sequence_from_ofsm_tables` in the implemented test strategies are replaced with a call to this function.

12.4 Containers Framework

The mechanised formalisation developed in Part III extensively employs the predefined data types `set` and `map`. As described above, these are, in the absence of further instructions to the code generator, implemented as lists and functions[6], respectively. On these representations, many standard operations are inefficient, such as checking whether a set contains a certain element, or not suitable for code generation, such as computing the domain of a map whose keys are of a type inhabited by infinitely many values.

The *Containers* framework presented in [70], available in the Archive of Formal Proofs as [69], provides several alternative implementations of sets and maps, as well as mechanisms for automatically selecting suitable implementations for the code generator based on type classes[7]. In executable implementations derived for the present work, sets and maps are thus implemented via red-black-trees, providing more efficient operations than lists. For sets, this replacement of implementations merely requires importing the framework into the formalisation, whereas for maps some data refinement is required, as the framework does not provide new code equations directly for type `map`, which is just a synonym for regular functions. Instead, the framework provides code equations for type `mapping`, and I provide in my formalisation code equations that replace usages of `map` with `mapping`. Appendix C exemplifies the use of `mapping` in providing efficient representations of prefix-closed sets of lists.

12.5 Generated Implementations

Final instructions to the code generator are provided in a separate theory file `Test_Suite_Generator_Code_Export.thy`. For each of the test strategy implementations discussed in Section 11.3, this theory file defines two new functions, returning different representations of test suites and introducing the optional step of first transforming the reference FSM into a language-equivalent minimal observable FSM[8]. These functions also fix the types of states, in-

[6] More precisely, type `('a,'b) map` is a synonym for the type `('a ⇒ 'b option)` of functions that map each key either to `None`, in which case the map contains no value for that key, or to some `(Some y)`, in which case the map contains value `y` for the key.

[7] In [67], Lammich et al. discuss data refinement and compare the *Containers* framework to the complementary but more heavy-weight *Autoref* framework (see [64]).

[8] This transformation is performed via function `to_prime` defined in theory file `Prime_Transformation.thy`, which first transforms a given FSM into a language-

puts, and outputs as `integer`, which are translated during code generation to arbitrary-precision integer types of the target language[9]. For example, from `w_method_via_h_framework` the following two functions are derived:

```
definition w_method_via_h_framework_ts ::
   "(integer,integer,integer) fsm ⇒
    integer ⇒
    bool ⇒
    ((integer×integer)×bool) list list"

definition w_method_via_h_framework_input ::
   "(integer,integer,integer) fsm ⇒
    integer ⇒
    bool ⇒
    integer list list"
```

Both functions expect three parameters: a reference FSM M, the upper bound on the additional number k of states of considered implementations, and a flag whether the reference model is already minimal, observable, and containing only reachable states. If this flag is not set, then both functions transform the reference model. Thereafter, a test suite TS is computed for fault domain $\mathcal{F}(M, m)$ using $m = |M| + k$. The functions differ only in the representation of the returned test suite. Functions with suffix `_ts` extend each IO-pair in each trace of TS with a Boolean flag (using abbreviated values `T`,`F`) indicating whether in M the pair is in the language of the state reached by the preceding pairs. For example, some $(x_1/y_1).(x_2/y_2).(x_3/y_3) \notin \mathcal{L}(M)$ with $(x_1/y_1).(x_2/y_2) \in \mathcal{L}(M)$ would result in augmented trace $(x_1/y_1, \mathtt{T}).(x_2/y_2, \mathtt{T}).(x_3/y_3, \mathtt{F})$. This augmentation encodes information on the containment in $\mathcal{L}(M)$ of prefixes of traces in TS and hence M itself is no longer required when the resulting test suite is applied to an SUT.

In contrast, functions with suffix `_input` return the input portions of traces in TS, which reduces the space required to store test suite, but requires responses of the SUT observed in testing to be compared against the behaviour of the reference model M. Furthermore, the reduction to input sequences loses information on opportunities to stop application of some test case $\bar{x}/\bar{y}$ early, as testing has to consider all responses to prefixes $\bar{x}$, not all of which are necessarily included in TS (see Appendix B). In both cases, I have chosen to represent the resulting test suite as a list of lists instead of a set of lists, in order to simplify the handling of test suites in the tools presented in Chapter 13.

Finally, `Test_Suite_Generator_Code_Export.thy` generates code for function `fsm_from_list` (introduced in Subsection 8.2.1) to facilitate creation of values of type `fsm`, as well as for several functions describing transformations of FSMs. Test strategies for reduction testing (see Appendix A) are also exported.

---

equivalent observable FSM via `make_observable` (see Section 8.4), which is then minimised via `minimise` (see Section 8.5) and finally restricted to its reachable states via `restrict_to_reachable_states` of `FSM.thy`.

[9] For Haskell and SML, this results in types `Integer` and `IntInf.int`, respectively.

Chapter 13

Toolset

In order to apply the test strategies implemented in Part III to practical systems under test, I provide a pair of small tools consisting of a test suite generator and a test harness that applies generated test suites to a given SUT. These are introduced in Sections 13.1 and 13.2, respectively. Thereafter, Section 13.3 discusses the benefits of employing code generated from Isabelle/HOL with respect to tool qualification. The subsequent Chapter 14 provides an evaluation of the test strategy implementations on randomly generated FSMs and compares observed results with those of hand-written implementations. The described tools, evaluation data, and further examples are available in the repository accompanying the present work (see Section 1.4). Fig. 13.1 visualises the components of this tool set and how they are employed to test an SUT against a given reference model.

13.1 Test Suite Generator

The generator is implemented as a command line tool `test-suite-generator` written in Standard ML, which imports the generated code described in Section 12.5. Using these imports, the tool generates a test suite for a given reference model M, a given strategy identified by some name (for example using string `H_Pair` to select `h_method_via_pair_framework`), a flag indicating whether M is already prime (i.e. observable, minimal, and containing only reachable states), and a number $k \in \mathbb{N}$ specifying the upper bound on additional states in the fault domain.

The use of provably correct implementations sets this test suite generator apart from other libraries and tools for testing based on FSMs, such as the `JPlavisFSM`[1] library (see [92]), the `FSMlib`[2] library (see [107]), the `libfsmtest`[3]

[1] https://github.com/adenilso/jplavisfsm
[2] https://github.com/Soucha/FSMlib
[3] https://bitbucket.org/JanPeleska/libfsmtest

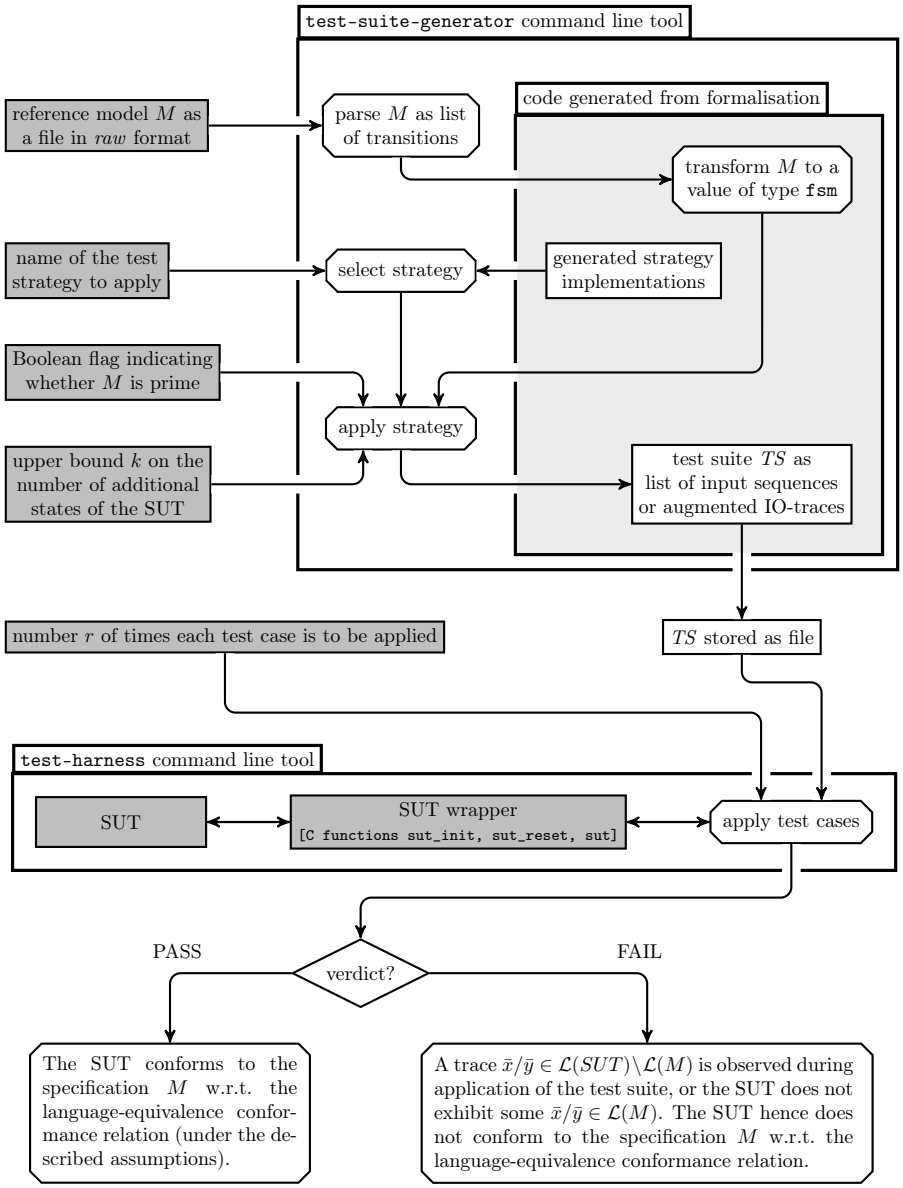


Figure 13.1: Overview of the workflow of using the `test-suite-generator` and `test-harness` command line tools to test an SUT. Dark grey shadings indicate inputs and SUT integration provided by the user, while light grey shadings indicate the trustworthy code generated from the Isabelle/HOL formalisation.

library (see [4]), and the `FSMTest`[4] tool (see [99]). Section 4 of [4] provides a short overview and comparison of these tools, each of which exhibits unique capabilities, such as mutation testing in `JPlavisFSM`, learning of FSMs in `FSMlib`, property-oriented testing in `libfsmtest`, and handling of extended finite state machines (see [31]) with `FSMTest`.

The test suite generator employs as format for encoding FSMs the so-called *raw* format of the open source `libfsmtest` library (see [4]), which for each transition (q, x, y, q') contains a single line `q x y q'`. The source state of the first listed transition specifies the initial state of the FSM. This is exemplified in Fig. 13.2. Only integers are employed as identifiers[5].

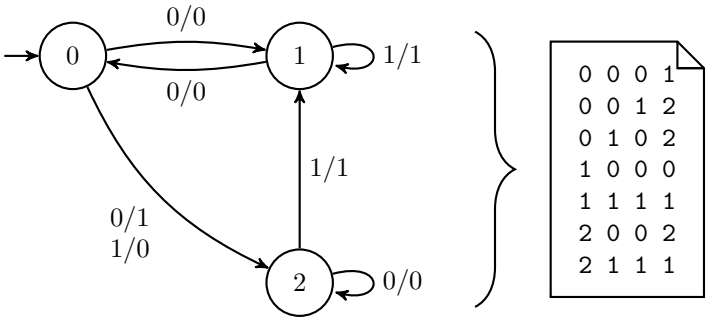


Figure 13.2: FSM M_{13} and its encoding in the *raw* format.

The generated test suite is encoded as described in Section 12.5. That is, it is either a list of input sequences or a list of IO-traces augmented with flags indicating language containment in the reference model. Applying the generator to prime FSM M_{13} as shown in Fig. 13.2 for $k = 0$ and strategy `H_Pair` results in input sequences $\{000, 0100, 1000, 1100\}$ or the following augmented IO-traces:

```
((0/0),T).((0/0),T).((0/1),T)
((0/0),T).((0/1),F)
((0/0),T).((1/0),F)
((0/0),T).((1/1),T).((0/0),T).((0/1),T)
((0/0),T).((1/1),T).((0/1),F)
((0/1),T).((0/0),T).((0/1),F)
((0/1),T).((0/1),F)
((1/0),T).((0/0),T).((0/0),T).((0/1),F)
((1/0),T).((0/0),T).((0/1),F)
((1/0),T).((0/1),F)
((1/0),T).((1/0),F)
((1/0),T).((1/1),T).((0/0),T).((0/1),T)
((1/0),T).((1/1),T).((0/1),F)
((1/1),F)
```

[4] http://fsmtestonline.ru

[5] Since all components of a well-formed finite state machine are finite and hence enumerable, this does not constitute a restriction.

13.2 Test Harness

The application of test suites to an SUT is performed via the `test-harness` command line tool, which interfaces with the SUT via three SUT-specific functions `sut_init()`, `sut_reset()`, and `sut()` implemented in an *SUT wrapper*[6]. Function `sut_init()` is called once before applying any test cases in order to set up the SUT, for example by initialising database connections. Next, `sut_reset()` is called before each application of a test case, in order to reset the SUT to its initial state. Finally, `sut()` applies a single input to the SUT and observes an output. Both the input supplied to `sut()` and the returned output are integers, so that `sut()` is furthermore responsible for translating inputs in the test suite to concrete inputs to the SUT, which may require more complex data types than integers, as well as for translating the observed output to an integer representing an output of the reference model. The SUT wrappers employed in `test-harness` are equivalent to those employed by the `libfsmtest` library (see [4]), which encode inputs in test suites as strings. The user manual of this library provides examples for the translation of inputs and outputs required by `sut()`.

As the `libfsmtest` library already supports application of test suites represented as lists of input sequences, the `test-harness` command line tool only considers test cases represented as augmented IO-traces. An test case

$$((x_1/y_1), b_1).((x_2/y_2), b_2).((x_3/y_3), b_3).\ldots.((x_k/y_k), b_k)$$

is applied by stepwise consideration of elements $((x_i/y_i), b_i)$ of the list, beginning at $i = 1$. For each $1 \leq i \leq k$, input x_i is applied to the SUT via `sut(`x_i`)`, observing a response y'_i. Application of the test suite stops as soon as one of the following conditions applies, which are checked in the given order:

1. If b_i = `T` and $y_i \neq y'_i$, then the current application of the test case is inconclusive, as it has not been shown that the SUT exhibits behaviour x_i/y_i in the state reached by preceding IO-pairs[7].
2. If b_i = `F` and $y_i = y'_i$, then the SUT fails the test case application by exhibiting a response not exhibited by the reference model.
3. If b_i = `F` and $y_i \neq y'_i$, then the SUT passes the test case application, as it has exhibited the IO-behaviour of the preceding IO-pairs (whose flags b_j must all be `T`) and not exhibited behaviour x_i/y_i in the state reached by preceding IO-pairs.
4. If $i = k$, then the SUT passes the test case. This conditions can occur only if all flags are `T` and for each $1 \leq j \leq k$ it holds that $y_j = y'_j$, implying that the SUT exhibits $x_1 \ldots x_k/y_1 \ldots y_k \in \mathcal{L}(M)$.

[6] The `test-harness` command line tool is written in Haskell and but expects the SUT wrapper to supply C functions – accessed via the foreign function interface of Haskell – in order to simplify integration.

[7] A more sophisticated application of test cases might check whether there exist another test case that coincides with the current test case in its first $(i-1)$ steps and whose i-th input is also x_i, and then try to continue execution of that test case.

The SUT fails the test case if any application of it fails or if all applications are inconclusive. Otherwise the SUT passes the test case. That is, the SUT passes a test case if and only if for all prefixes $((x_1/y_1), b_1).....((x_k/y_k), \texttt{F})$ of the test case it holds that $x_1 \ldots x_k/y_1 \ldots y_k \notin \mathcal{L}(M)$ is not exhibited by the SUT, while for all prefixes $((x_1/y_1), \texttt{T}).....((x_j/y_j), \texttt{T})$ augmented only with `T` it holds that the SUT does exhibit $x_1 \ldots x_j/y_1 \ldots y_j \in \mathcal{L}(M)$[8]. A test suite is passed by the SUT if and only if all test cases in the test suite are passed by the SUT.

As an example, consider an SUT that introduces a transition fault into M_{13} (Fig. 13.2) by redirecting the transition $(2, 1, 1, 1)$ to target state 0. This failure is uncovered by the second to last test case of the test suite developed in the previous section, `((1/0),T).((1/1),T).((0/1),F)`, as (1/0).(1/1).(0/1) is in the language of the SUT. The failure is also uncovered via the preceding test case `((1/0),T).((1/1),T).((0/0),T).((0/1),T)`, as (1/0).(1/1).(0/0).(0/1) is not in the language of the SUT and hence all applications of this test case are inconclusive.

Note that observing all responses of a nondeterministic SUT may require applying test cases multiple times. For nondeterministic SUTs, I thus employ the so-called *complete testing assumption* (see Subsection 3.2.2), which assumes that there exists some r such that each response of the SUT to some $\bar{x}$ can be observed by applying $\bar{x}$ at most r times. This r constitutes the final parameter of `test-harness` (see Fig. 13.1). If the SUT is deterministic, then any inconclusive test case application is a failed test case application, since it indicates that for some prefix $((x_1/y_1), \texttt{T}).....((x_j/y_j), \texttt{T})$ of the test case a response to input sequence $x_1 \ldots x_j$ other than the single response $y_1 \ldots y_j$ of the reference model has been observed in the SUT.

13.3 Tool Qualification

The two tools described above each contain fewer than 200 lines of code in addition to the generated code and serve only as interfaces between the reference model, test suites and the SUT, with most lines of code handling parsing. They can thus be verified manually with little effort. Together with the trustworthiness of the provably correct generated code (see [42]) facilitating the test suite generator, the generation of implementations from proven mechanised formalisations provides significant benefits for tool qualification efforts of test tools that employ the above tools: First, the fault detection capabilities of generated test suites are established by mechanised proofs within the formalisation, providing evidence that they are complete for the fault domain of SUTs whose behaviour can be represented by an OFSM that contains at most k states more than the prime reference model and has the same set of inputs and outputs as the latter. Verification of these proofs in turn reduces to verifying the used definitions, as any proof is verified automatically by the small trustworthy inference kernel of

[8] As languages of FSMs are prefix closed, test cases thus need only be applied until the first IO-pair augmented with `F`. The implementations described Section 11.3 generate only test cases that contain at most a single `F`, located in the last element.

Isabelle, even when using sophisticated automated proof methods (see [8]). The qualification of formal method tools has been discussed in [17, 24, 117].

In [17], Brauer et al. identify two particular hazards to tool qualification that may be introduced by malfunctions of test tools that generate and apply tests to an SUT, such as the pair of tools described above. First, the tools may fail to detect deviations of the behaviour of the SUT from that of the reference model, if the application of test cases is implemented erroneously and fails to recognise or report such deviations. Second, the tools may fail to ensure satisfaction of pre-conditions of test cases, in which case the result of applying the test case may not reflect the intention of the test case.

For the above pair of tools and test cases represented as augmented IO-traces, mitigation of the first hazard – undetected SUT failures – reduces to verifying that the test case application implemented in the `test-harness` tool conforms to the description given in Section 13.2, as the behaviour of the reference model is encoded in the test case itself. In contrast, if test cases are represented as input sequences $\bar{x}$, then the responses of the reference model to $\bar{x}$ are obtained via simulation of the reference model and then compared to the observed responses of the SUT, which introduces more components of the test harness to be verified[9]. For both representations of test suites, the responses of the SUT to test cases and the derived verdict are logged.

Mitigation of the second hazard – unsatisfied pre-conditions – is for the above pair of tools limited to verifying that the `sut_reset()` function of the SUT wrapper reliably resets the SUT to its initial state, as this constitutes the only pre-condition on test cases for the complete test strategies discussed in the present work. Since the implementation of the SUT wrapper is highly dependent on the SUT, I have not included the test harness in the Isabelle formalisation, but I believe that the small provided implementation can easily be verified manually for a given SUT[10].

The integration of code generated from provably correct formalisations into the tool set, together with encoding the behaviour expected by the reference model into test cases, thus obviates measures such as the replay of observed behaviours against the specification (see [17]) arising from untrusted test case generators, and reduces the size of manually written components of the tool set, reducing the effort required for manual verification.

[9] Note that theory file `FSM.thy` provides functions to compute the responses of an FSM to some input sequence $\bar{x}$. Thus, by generating code for this function, errors in manually written functions to simulate the reference model may be avoided.

[10] Naturally, this verification should also cover functions `sut_init()` and `sut()`, since errors in these functions may result in test cases being applied incorrectly or to an SUT not set up in a well-specified state.

Chapter 14

Experiments

I have applied the test suite generator described in Section 13.1 to sets of randomly generated FSMs in order to evaluate the size of obtained test suites and to compare the performance of the generated code to manually written implementations. The sets of FSMs and the obtained observation data described below are publicly available at the repository accompanying this work (see Section 1.4). In the following, I examine only prime deterministic FSMs and consider test suites represented as sets of input sequences, in order to allow direct comparison with previous efforts to evaluate and compare test strategies, such as [26] and [29], which employ the same restrictions.

14.1 Comparison of Implemented Strategies

Fig. 14.1 shows the result of applying several test strategies to randomly generated deterministic FSMs M with 2 inputs and 2 outputs, using $k = 0$ (that is, generating test suites that are $|M|$-complete). Here, methods using the H and Pair-Frameworks are denoted with suffix or infix `_H` and `_Pair`, respectively. Identifiers `H_Pair_3_11` and `SPYH_H_10` additionally encode the two Boolean arguments passed to functions `h_method_via_pair_framework_3` and `spyh_method_via_h_framework` in their last two characters and hence express the usage of particular completions (see Chapter 11). In Fig. 14.1, I omit other combinations of Boolean arguments for these two strategies, as these obtain larger test suites. The results for the W-Method are shown only partially due to the rapid growth of the generated test suite.

The observed values in Fig. 14.1 indicate that, on average, for the given DFSMs the following relations hold on the average sizes of obtained test suites (measured as the number of maximal contained input sequences):

$$T_W > T_{Wp} > T_{HSI} > T_H > T_{SPY} > T_{H3} > T_{SPYH}$$

where T_S denotes the average size of test suites obtained by test strategy S, measured as the number of maximal input sequences. Note that the H-Method

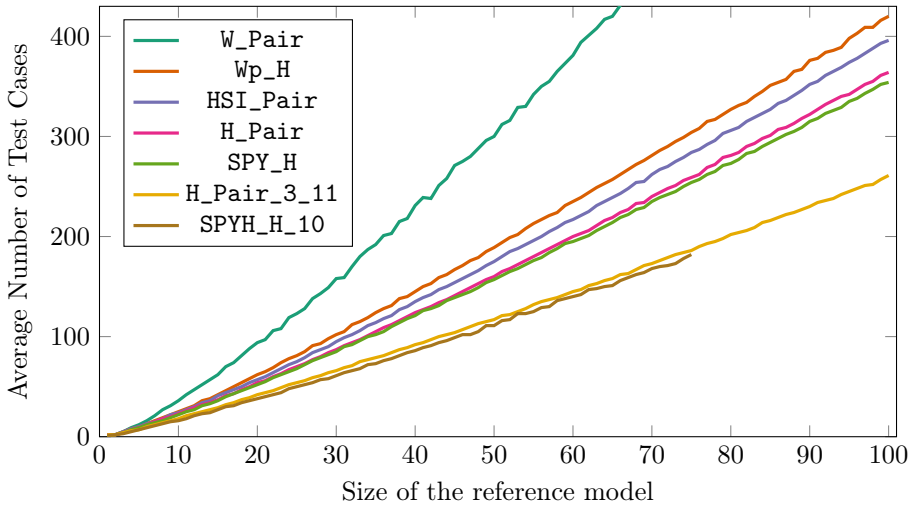


Figure 14.1: Average number of test cases (maximal input sequences) obtained by applying test strategies to 100 randomly generated prime DFSMs M with 2 inputs and 2 outputs for each $1 \leq |M| \leq 100$ using $k = 0$.

occurs twice, as two distinct implementations have been considered here: H (`H_Pair`) and H^3 (`H_Pair_3_11`). Table 14.1 shows some values for T_W not depicted in Fig. 14.1 due to the rapid growth of test suites generated by the W-Method. Values for T_{SPYH} are omitted after $|M| = 77$ due to time constraints[1].

The relations observed in Fig. 14.1 are in agreement with those observed in similar experiments in the literature: Dorofeeva et al. in [26] in particular observe $T_W > T_{Wp} > T_H$ and $T_W > T_{HSI} > T_H$, but little difference between T_{Wp} and T_{HSI} due to a more sophisticated implementation of the Wp-Method compared to `wp_method_via_h_framework` (see Algorithm 3). Simão et al. in [103] observe $T_{HSI} > T_{SPY}$. Finally, Endo et al. in [29] observe $T_W > T_{HSI} > T_H > T_{SPY}$, whereas Soucha et al. in [108] and Soucha in [106] observe $T_W > T_{Wp} > T_{HSI} > T_{SPY} > T_H > T_{SPYH}$, thus differing in the observations for the H and SPY-Methods. As exemplified by $T_H > T_{SPY} > T_{H3}$ in Fig. 14.1, such divergent observations may result from applying distinct implementations of the same test strategies. In [106] it is observed that the S-Method generates even smaller test suites than the SPYH-Method[2].

[1] At $|M| = 77$ and larger reference models, the average time required to compute the test suite for a single DFSM in the data set exceeds 300s. This is a result of the simple implementation of convergence graph `simple_cg` (see Section 9.4), which the SPYH-Method employs extensively.

[2] On the data set applied here, the results of the $S_{partial}$-Method differ little from the SPYH-Method and hence have been omitted in the discussion and Fig. 14.1.

Table 14.1: Average test suite sizes as shown in Fig. 14.1 for selected sizes n of the reference model.

n	T_W	T_{Wp}	T_{HSI}	T_H	T_{SPY}	T_{H3}	T_{SPYH}
25	123	81	75	70	68	54	50
50	300	189	175	160	157	117	111
75	505	304	284	259	254	186	182
100	719	420	396	364	354	261	-

14.2 Comparison with a Manually Developed Implementation

Many benefits of employing provably correct implementations generated from the Isabelle formalisation discussed in Section 13.3 are only practically relevant if these implementations are sufficiently efficient to be applicable to non-trivial inputs. In order to evaluate the generated implementations, I compare the two implementations of the H-Method employed in the previous section – `H_Pair` and the more elaborate `H_Pair_3_11` – against a manually developed implementation of the H-Method in the open source C++ library `fsmlib-cpp`[3] that has been employed, for example, in [47][4]. I have chosen to consider the H-Method for this comparison, as it performs the most complex heuristics of strategies that do not exploit convergence. As discussed in Section 9.4, the current implementation of convergence graph STANDARD is very simple, limiting the performance of implementations of the SPY and SPYH-Methods.

Fig. 14.2 shows test suite sizes and average computation times for the three implementations of the H-Method over randomly generated prime DFSMs with 3 inputs, 3 outputs, and up to 80 states, considering the same set of DFSMs for each $k \in \{0, 1, 2\}$. The computation times have been measured on a machine running Ubuntu 18.04 equipped with an Intel Core i7-4700MQ processor and 16GB of RAM, employing MLton 20210117[5] to compile the SML code generated from Isabelle/HOL and GCC 9.4[6] to compile the `fsmlib-cpp`. Table 14.2 lists selected values of Fig. 14.2 for a more precise comparison.

As shown in Fig. 14.2, for all $k \in \{0, 1, 2\}$, the test suites generated using function `H_Pair` (`h_method_via_pair_framework`) are on average larger than those obtained using the `fsmlib-cpp`, which are in turn larger than those computed via `H_Pair_3_11` (`h_method_via_pair_framework_3` with both completion flags set to `True`). With respect to the time required to compute test

[3] Publicly available at https://github.com/agbs-uni-bremen/fsmlib-cpp. The H-Method considered here is available as method `hMethodOnMinimisedDfsm` of class `Dfsm`.

[4] Note that the `fsmlib-cpp` library has been superseded by the open source `libfsmtest` C++ library introduced in [4]. The implementation of the H-Method in this library, however, differs significantly in its heuristics from those employed in the `fsmlib-cpp` library and my formalisation, and is, at the time of writing, in the process of being thoroughly redesigned.

[5] http://mlton.org/Release20210117

[6] https://gcc.gnu.org/releases.html

Table 14.2: Average test suite sizes T and computation times t (in s) as shown in Fig. 14.2 for selected sizes n of the reference model and $k \in \{0, 1, 2\}$.

k	n	T			t		
		`H_Pair`	`fsmlib-cpp`	`H_Pair_3_11`	`H_Pair`	`fsmlib-cpp`	`H_Pair_3_11`
0	20	110	73	74	0.01	0.04	0.04
0	40	248	176	160	0.06	0.20	0.32
0	60	399	297	252	0.16	0.53	1.31
0	80	554	429	346	0.32	1.09	3.53
1	20	325	199	183	0.02	0.25	0.12
1	40	735	459	408	0.13	1.34	1.13
1	60	1173	746	662	0.30	3.71	4.60
1	80	1634	1051	917	0.58	7.43	12.28
2	20	977	567	523	0.08	2.15	0.40
2	40	2205	1249	1077	0.36	11.34	4.08
2	60	3520	2003	1681	1.03	30.86	18.41
2	80	4904	2786	2327	1.73	62.90	51.20

suites, the behaviour of the implementations varies more significantly depending on both k and the size of the reference model. In general, `H_Pair` requires less time to compute test suites, which may be explained by its significantly less elaborate heuristics, resulting in larger test suites. This difference in speed increases significantly with increasing k. For $k = 0$, the implementation of the `fsmlib-cpp` computes test suites significantly faster than `H_Pair_3_11`. For $k = 1$, this occurs only after an observed threshold of $|M| = 50$, with both functions exhibiting comparable times for $|M| < 50$. For $k = 2$, the `fsmlib-cpp` is slower than `H_Pair_3_11` over all considered $1 \leq |M| \leq 80$, indicating that the `fsmlib-cpp` is more sensitive to k than `H_Pair_3_11` on small FSMs.

In summary, for certain inputs the implementations generated from the formalisation in Isabelle/HOL are capable of generating smaller test suites in less time than the manually developed implementation of the `fsmlib-cpp`. Furthermore, Fig. 14.2 indicates that `H_Pair` may be able to generate test suites for values of k or sizes of the reference model where application of the implementation provided in the `fsmlib-cpp` library is not feasible due to the time required. By more extensive program and data refinement, these results can be improved further. Note here also that the implementation of the H-Method in the `fsmlib-cpp` library is restricted to complete deterministic FSMs, which allows simplification of various operations on FSMs, and the use of test suites represented as inputs during computation without exhibiting the problems discussed in Appendix B. Functions `H_Pair` and `H_Pair_3_11` impose no such restrictions. Specialised handling of DFSMs may be implemented in future work as another example for program refinement.

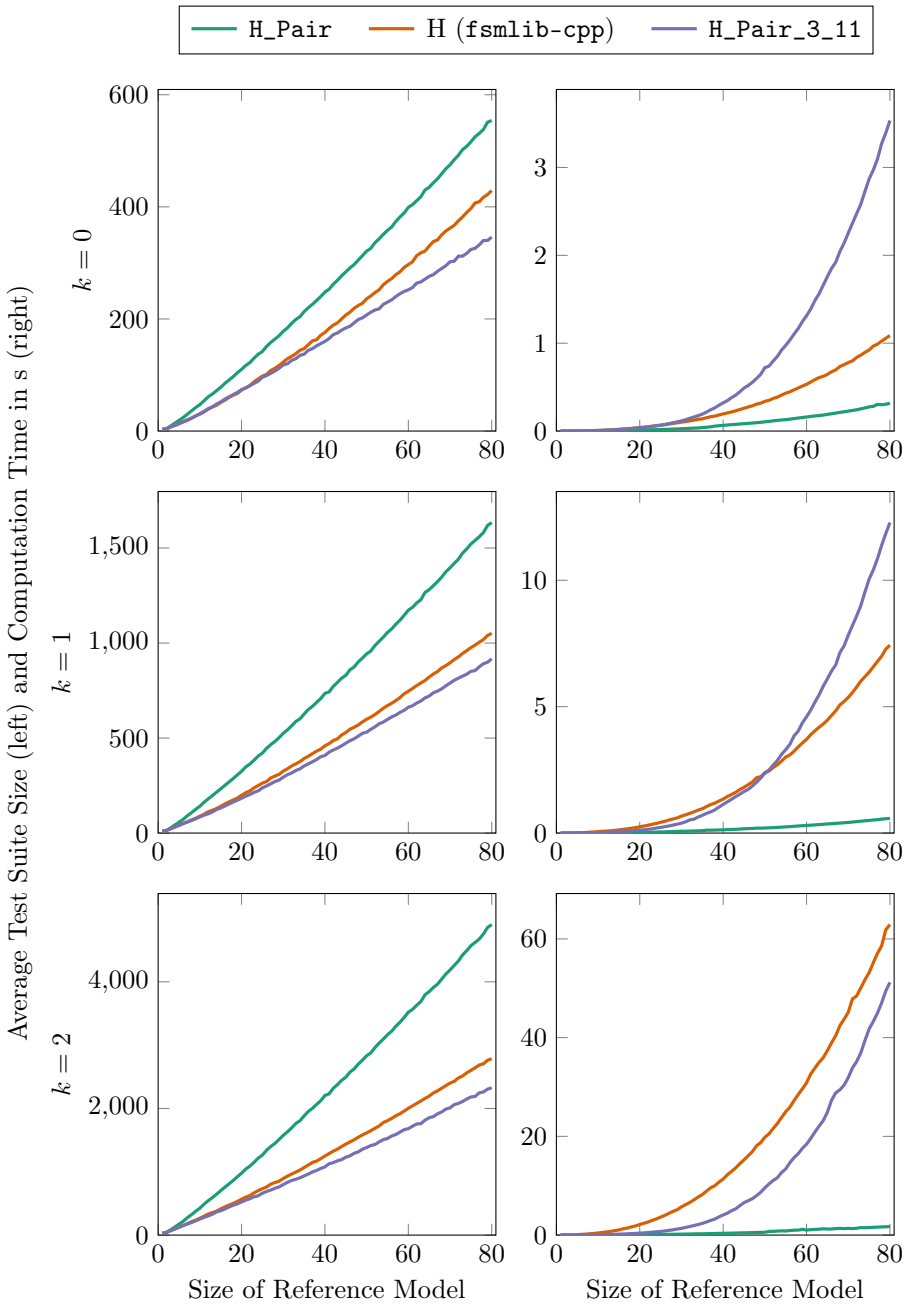


Figure 14.2: Average number of test cases (maximal input sequences) obtained by applying test strategies to 100 randomly generated prime DFSMs M with 3 inputs and 3 outputs for each $1 \leq |M| \leq 80$, using $k \in \{0, 1, 2\}$.

Part V

Conclusions and Future Work

Chapter 15

Conclusions

This dissertation describes the development of a comprehensive mechanised formalisation of several complete test strategies for the language-equivalence conformance relation on finite state machines using the proof assistant Isabelle. The formalisation effort entails three main contributions.

First, the presentation and implementation of the strategies has been unified in Chapter 6 by providing generic frameworks such that the various strategies can be decomposed into independent functions to be passed to the frameworks. By implementing multiple variants of the W-Method (see [23, 115]) and H-Method (see [27]), as well as by adding a partial implementation of the S-Method introduced in [106], it has been shown that these frameworks support the implementation of new variations of the considered strategies, while reducing the effort in implementing these variations by allowing the reuse of steps shared with previously implemented strategies.

Next, the completeness of the considered strategies is established in Part III by mechanised proofs based on the frameworks. Via the provision of suitable interface lemmata, the completeness proofs are decoupled from concrete implementations of the arguments passed to the generic frameworks. This reduces the task of proving complete a particular strategy to proving that the arguments it passes to a framework satisfy certain properties. That is, if a new strategy differs only in a single argument from a previously implemented strategy, then only the properties of this argument need to be established in order to prove completeness.

Finally, from the implementation of a strategy via a framework and the corresponding completeness proof, it is possible to generate a provably complete executable implementation of that strategy to be employed in test tools. Part IV shows how efficient implementations can be obtained via program and data refinement, resulting in implementations that are reasonably efficient compared to manually developed implementations. At the same time, the use of provably correct implementations mitigates many hazards to tool qualification introduced by untrusted test suite generators.

Chapter 16

Future Work

A natural future research topic based on the present work is the extension of the formalisation to cover further strategies, conformance relations, and modelling formalisms. My previous work [95] already covers a strategy for reduction testing, which can be extended to realise the strategies for testing quasi-reduction and quasi-equivalence developed in [43, 45] or the strategy for the strong reduction conformance relation introduced in [97]. Furthermore, the Pair-Framework developed in Section 6.5 has been designed in order to support variations of the H-Method such as the safety-complete H-Method introduced in [47], the requirements-based test strategy developed in [49], or the application to property oriented testing introduced in [60]. Note that these strategies all employ abstractions of the reference model, which can also be formalised. In the case of [60], this would require introduction of symbolic FSMs (SFSMs).

As discussed in Section 14.2, more elaborate implementations of strategies such as the H, SPY, and SPYH-Method currently result in generated implementations that are slower than manually developed implementations for certain inputs. It would be of interest to see how the performance of the generated implementations might be improved by further program and data refinement, in particular by implementing a more sophisticated representation of convergence graphs such as that employed in [106], or by employing more imperative features in the refinement (see [66]).

The formalisation of complete test strategies from which to generate code is not the only possible application of proof assistants in the generation of provably complete test suites. Another approach is to formalise and generate a provably correct tool that is able to verify completeness of a given test suite for a given reference model. Such a tool might be based on the algorithms to check completeness of test suites developed in [12, 13, 14, 100, 116]. Alternatively, the tool could be restricted to a certain test strategy and require as additional parameters certain information on the steps taken in the construction of the test suite. Consider, for example, the task of checking whether some test suite TS generated by an untrusted implementation of the W-Method is complete for given reference model M and fault domain $\mathcal{F}(M, m)$. If this implementation of

the W-Method additionally generates witnesses V and W of the supposed state cover and characterisation set used in generating the test suite, the completeness check may be simplified significantly. It requires only checking whether V and W are valid for M, and whether TS contains $\{\alpha\}.W$ for all relevant traces α, which depend on V, M and m. This approach allows arbitrarily complex algorithms and data structures to be employed in the generation of the test suite, while still obtaining a trustworthy verdict on the completeness of the obtained test suite. Compared to the fully formalised test strategy implementations developed in the present work it does not, however, guarantee completeness of the generated test suite. In [38], Gleirscher et al. employ a manually developed *test suite validator* for the H-Method implemented in the `libfsmtest` library (see [4]). Provably correct checkers have already been developed using Isabelle/HOL for other domains, for example to check SAT certificates (see [65]) or proof certificates for algebraic reasoning (see [55]).

Bibliography

[1] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, Artem Starostin, and Alexandra Tsyban. "Balancing the Load". In: *J. Autom. Reason.* 42.2-4 (2009), pp. 389–454. DOI: 10.1007/s10817-009-9123-z. URL: https://doi.org/10.1007/s10817-009-9123-z.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. "An orchestrated survey of methodologies for automated software test case generation". In: *J. Syst. Softw.* 86.8 (2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061. URL: https://doi.org/10.1016/j.jss.2013.02.061.

[3] David Aspinall and Cezary Kaliszyk. "Towards Formal Proof Metrics". In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Perdita Stevens and Andrzej Wasowski. Vol. 9633. Lecture Notes in Computer Science. Springer, 2016, pp. 325–341. DOI: 10.1007/978-3-662-49665-7_19. URL: https://doi.org/10.1007/978-3-662-49665-7%5C_19.

[4] Moritz Bergenthal, Niklas Krafczyk, Jan Peleska, and Robert Sachtleben. "libfsmtest An Open Source Library for FSM-Based Testing". In: *Testing Software and Systems*. Ed. by David Clark, Hector Menendez, and Ana Rosa Cavalli. Cham: Springer International Publishing, 2022, pp. 3–19. ISBN: 978-3-031-04673-5.

[5] Nikolaj Bjørner. "Z3 and SMT in Industrial R&D". In: *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. Ed. by Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink. Vol. 10951. Lecture Notes in Computer Science. Springer, 2018, pp. 675–678. ISBN: 978-3-319-95581-0. DOI: 10.1007/978-3-319-95582-7_44. URL: https://doi.org/10.1007/978-3-319-95582-7%5C_44.

[6] Jasmin Christian Blanchette. *Hammering Away*. 2021. URL: https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf (visited on 2022-04-05).

[7] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers". In: *Journal of Automated Reasoning* 51.1 (2013), pp. 109–128. DOI: 10.1007/s10817-013-9278-5. URL: https://doi.org/10.1007/s10817-013-9278-5.

[8] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. "Automatic Proof and Disproof in Isabelle/HOL". In: *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*. Ed. by Cesare Tinelli and Viorica Sofronie-Stokkermans. Vol. 6989. Lecture Notes in Computer Science. Springer, 2011, pp. 12–27. DOI: 10.1007/978-3-642-24364-6_2. URL: https://doi.org/10.1007/978-3-642-24364-6%5C_2.

[9] Jasmin Christian Blanchette, Max W. Haslbeck, Daniel Matichuk, and Tobias Nipkow. "Mining the Archive of Formal Proofs". In: *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*. Ed. by Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge. Vol. 9150. Lecture Notes in Computer Science. Springer, 2015, pp. 3–17. DOI: 10.1007/978-3-319-20615-8_1. URL: https://doi.org/10.1007/978-3-319-20615-8%5C_1.

[10] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 93–110. DOI: 10.1007/978-3-319-08970-6_7. URL: https://doi.org/10.1007/978-3-319-08970-6%5C_7.

[11] Sascha Böhme and Tobias Nipkow. "Sledgehammer: Judgement Day". In: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. Lecture Notes in Computer Science. Springer, 2010, pp. 107–121. DOI: 10.1007/978-3-642-14203-1_9. URL: https://doi.org/10.1007/978-3-642-14203-1%5C_9.

[12] Adilson Luiz Bonifácio and Arnaldo Vieira Moura. "On the completeness of test suites". In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. by Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong. ACM, 2014, pp. 1287–1292. DOI: 10.1145/2554850.2554968. URL: https://doi.org/10.1145/2554850.2554968.

[13] Adilson Luiz Bonifácio and Arnaldo Vieira Moura. "Test suite completeness and black box testing". In: *Software Testing, Verification and Reliability* 27.1-2 (2017). DOI: 10.1002/stvr.1626. URL: https://doi.org/10.1002/stvr.1626.

[14] Adilson Luiz Bonifácio, Arnaldo Vieira Moura, and Adenilso da Silva Simão. "Experimental comparison of approaches for checking completeness of test suites from finite state machines". In: *Information and Software Technology* 92 (2017), pp. 95–104. DOI: 10.1016/j.infsof.2017.07.012. URL: https://doi.org/10.1016/j.infsof.2017.07.012.

[15] Ralph Bottesch, Max W. Haslbeck, Alban Reynaud, and René Thiemann. "Verifying a Solver for Linear Mixed Integer Arithmetic in Isabelle/HOL". In: *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*. Ed. by Ritchie Lee, Susmit Jha, and Anastasia Mavridou. Vol. 12229. Lecture Notes in Computer Science. Springer, 2020, pp. 233–250. DOI: 10.1007/978-3-030-55754-6_14. URL: https://doi.org/10.1007/978-3-030-55754-6%5C_14.

[16] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. "Challenges and Experiences in Managing Large-Scale Proofs". In: *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*. Ed. by Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge. Vol. 7362. Lecture Notes in Computer Science. Springer, 2012, pp. 32–48. DOI: 10.1007/978-3-642-31374-5_3. URL: https://doi.org/10.1007/978-3-642-31374-5%5C_3.

[17] Jörg Brauer, Jan Peleska, and Uwe Schulze. "Efficient and Trustworthy Tool Qualification for Model-Based Testing Tools". In: *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*. Ed. by Brian Nielsen and Carsten Weise. Vol. 7641. Lecture Notes in Computer Science. Springer, 2012, pp. 8–23. DOI: 10.1007/978-3-642-34691-0_3. URL: https://doi.org/10.1007/978-3-642-34691-0%5C_3.

[18] Achim D. Brucker and Burkhart Wolff. "Interactive Testing with HOL-TestGen". In: *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*. Ed. by Wolfgang Grieskamp and Carsten Weise. Vol. 3997. Lecture Notes in Computer Science. Springer, 2005, pp. 87–102. DOI: 10.1007/11759744_7. URL: https://doi.org/10.1007/11759744%5C_7.

[19] Achim D. Brucker and Burkhart Wolff. "Monadic Sequence Testing and Explicit Test-Refinements". In: *Tests and Proofs - 10th International Conference, TAP@STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*. Ed. by Bernhard K. Aichernig and Carlo A. Furia. Vol. 9762. Lecture Notes in Computer Science. Springer, 2016, pp. 17–36. DOI: 10.1007/978-3-319-41135-4_2. URL: https://doi.org/10.1007/978-3-319-41135-4%5C_2.

[20] Achim D. Brucker and Burkhart Wolff. "On theorem prover-based testing". In: *Formal Aspects of Computing* 25.5 (2013), pp. 683–721. DOI: 10.1007/s00165-012-0222-y. URL: https://doi.org/10.1007/s00165-012-0222-y.

[21] Lukas Bulwahn. "The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 92–108. DOI: 10.1007/978-3-642-35308-6_10. URL: https://doi.org/10.1007/978-3-642-35308-6%5C_10.

[22] Wendy Y. L. Chan, Son T. Vuong, and M. Robert Ito. "An Improved Protocol Test Generation Procedure Based on UIOs". In: *Symposium Proceedings on Communications Architectures & Protocols*. SIGCOMM '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 283–294. ISBN: 0897913329. DOI: 10.1145/75246.75274. URL: https://doi.org/10.1145/75246.75274.

[23] Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* 4.3 (1978), pp. 178–187. DOI: 10.1109/TSE.1978.231496. URL: https://doi.org/10.1109/TSE.1978.231496.

[24] Darren D. Cofer, Gerwin Klein, Konrad Slind, and Virginie Wiels. "Qualification of Formal Methods Tools (Dagstuhl Seminar 15182)". In: *Dagstuhl Reports* 5.4 (2015), pp. 142–159. DOI: 10.4230/DagRep.5.4.142. URL: https://doi.org/10.4230/DagRep.5.4.142.

[25] Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. "A Verified Implementation of the Berlekamp-Zassenhaus Factorization Algorithm". In: *Journal of Automated Reasoning* 64.4 (2020), pp. 699–735. DOI: 10.1007/s10817-019-09526-y. URL: https://doi.org/10.1007/s10817-019-09526-y.

[26] Rita Dorofeeva, Khaled El-Fakih, Stéphane Maag, Ana R. Cavalli, and Nina Yevtushenko. "FSM-based conformance testing methods: A survey annotated with experimental evaluation". In: *Information and Software Technology* 52.12 (2010), pp. 1286–1297. DOI: 10.1016/j.infsof.2010.07.001. URL: https://doi.org/10.1016/j.infsof.2010.07.001.

[27] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. "An Improved Conformance Testing Method". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*. Ed. by Farn Wang. Vol. 3731. Lecture Notes in Computer Science. Springer, 2005, pp. 204–218. DOI: 10.1007/11562436_16. URL: https://doi.org/10.1007/11562436%5C_16.

[28] Rita Dorofeeva, Nina Yevtushenko, Khaled El-Fakih, and Ana R. Cavalli. "Experimental Evaluation of FSM-Based Testing Methods". In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*. Ed. by Bernhard K. Aichernig and Bernhard Beckert. IEEE Computer Society, 2005, pp. 23–32. DOI: 10.1109/SEFM.2005.17. URL: https://doi.org/10.1109/SEFM.2005.17.

[29] André Takeshi Endo and Adenilso da Silva Simão. "Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods". In: *Information and Software Technology* 55.6 (2013), pp. 1045–1062. DOI: 10.1016/j.infsof.2013.01.001. URL: https://doi.org/10.1016/j.infsof.2013.01.001.

[30] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. "A Fully Verified Executable LTL Model Checker". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 463–478. DOI: 10.1007/978-3-642-39799-8_31. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_31.

[31] Khaled El-Fakih, Anton Kolomeez, Svetlana Prokopenko, and Nina Yevtushenko. "Extended Finite State Machine Based Test Derivation Driven by User Defined Faults". In: *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*. IEEE Computer Society, 2008, pp. 308–317. DOI: 10.1109/ICST.2008.16. URL: https://doi.org/10.1109/ICST.2008.16.

[32] Khaled El-Fakih, Nina Yevtushenko, and Natalia Kushik. "Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation". In: *Formal Aspects of Computing* 30.2 (2018), pp. 319–332. DOI: 10.1007/s00165-017-0450-2. URL: https://doi.org/10.1007/s00165-017-0450-2.

[33] Khaled El-Fakih, Nina Yevtushenko, and Ayat Saleh. "Incremental and Heuristic Approaches for Deriving Adaptive Distinguishing Test Cases for Non-deterministic Finite-State Machines". In: *The Computer Journal* 62.5 (2019), pp. 757–768. DOI: 10.1093/comjnl/bxy086. URL: https://doi.org/10.1093/comjnl/bxy086.

[34] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. "A verified SAT solver with watched literals using imperative HOL". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. Ed. by June Andronick and Amy P. Felty. ACM, 2018, pp. 158–171. DOI: 10.1145/3167080. URL: https://doi.org/10.1145/3167080.

[35] Susumo Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokthar Amalou, and Abderrazak Ghedamsi. "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering* 17.6 (1991), pp. 591–603. ISSN: 0098-5589. DOI: 10.1109/32.87284. URL: https://doi.org/10.1109/32.87284.

[36] Zvi Galil and Giuseppe F. Italiano. "Data Structures and Algorithms for Disjoint Set Union Problems". In: *ACM Computing Surveys (CSUR)* 23.3 (1991), pp. 319–344. DOI: 10.1145/116873.116878. URL: https://doi.org/10.1145/116873.116878.

[37] Arthur Gill. *Introduction to the theory of finite-state machines*. New York: McGraw-Hill, 1962.

[38] Mario Gleirscher, Lukas Plecher, and Jan Peleska. "Sound Development of Safety Supervisors". In: *CoRR* abs/2203.08917 (2022). DOI: 10.48550/arXiv.2203.08917. arXiv: 2203.08917. URL: https://doi.org/10.48550/arXiv.2203.08917.

[39] Florian Haftmann. *Code generation from Isabelle/HOL theories*. 2021. URL: https://isabelle.in.tum.de/dist/doc/codegen.pdf (visited on 2022-04-05).

[40] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar*. 2021. URL: https://isabelle.in.tum.de/dist/doc/classes.pdf (visited on 2022-04-05).

[41] Florian Haftmann, Alexander Krauss, Ondrej Kuncar, and Tobias Nipkow. "Data Refinement in Isabelle/HOL". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 100–115. DOI: 10.1007/978-3-642-39634-2_10. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_10.

[42] Florian Haftmann and Tobias Nipkow. "Code Generation via Higher-Order Rewrite Systems". In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer, 2010, pp. 103–117. DOI: 10.1007/978-3-642-12251-4_9. URL: https://doi.org/10.1007/978-3-642-12251-4%5C_9.

[43] Robert M. Hierons. "FSM quasi-equivalence testing via reduction and observing absences". In: *Science of Computer Programming* 177 (2019), pp. 1–18. DOI: 10.1016/j.scico.2019.03.004. URL: https://doi.org/10.1016/j.scico.2019.03.004.

[44] Robert M. Hierons. "Testing from a Nondeterministic Finite State Machine Using Adaptive State Counting". In: *IEEE Transactions on Computers* 53.10 (2004), pp. 1330–1342. DOI: 10.1109/TC.2004.85. URL: https://doi.org/10.1109/TC.2004.85.

[45] Robert M. Hierons. "Testing from Partial Finite State Machines without Harmonised Traces". In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1033–1043. DOI: 10.1109/TSE.2017.2652457. URL: https://doi.org/10.1109/TSE.2017.2652457.

[46] John E. Hopcroft and Jeffrey D. Ullman. "Introduction to Automata Theory, Languages and Computation". In: (1979).

[47] Wen-ling Huang, Sadik Özoguz, and Jan Peleska. "Safety-complete test suites". In: *Software Quality Journal* 27.2 (2019), pp. 589–613. DOI: 10.1007/s11219-018-9421-y. URL: https://doi.org/10.1007/s11219-018-9421-y.

[48] Wen-ling Huang and Jan Peleska. "Complete model-based equivalence class testing for nondeterministic systems". In: *Formal Aspects Comput.* 29.2 (2017), pp. 335–364. DOI: 10.1007/s00165-016-0402-2. URL: https://doi.org/10.1007/s00165-016-0402-2.

[49] Wen-ling Huang and Jan Peleska. "Complete Requirements-based Testing with Finite State Machines". In: *CoRR* abs/2105.11786 (2021). arXiv: 2105.11786. URL: https://arxiv.org/abs/2105.11786.

[50] Felix Hübner, Wen-ling Huang, and Jan Peleska. "Experimental evaluation of a novel equivalence class partition testing strategy". In: *Software & Systems Modeling* 18.1 (2019), pp. 423–443. DOI: 10.1007/s10270-017-0595-8. URL: https://doi.org/10.1007/s10270-017-0595-8.

[51] Brian Huffman and Ondrej Kuncar. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 131–146. DOI: 10.1007/978-3-319-03545-1_9. URL: https://doi.org/10.1007/978-3-319-03545-1%5C_9.

[52] Lars Hupel and Tobias Nipkow. "A Verified Compiler from Isabelle/HOL to CakeML". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 999–1026. DOI: 10.1007/978-3-319-89884-1_35. URL: https://doi.org/10.1007/978-3-319-89884-1%5C_35.

[53] Joe Hurd. "First-order proof tactics in higher-order logic theorem provers". In: *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports* (2003), pp. 56–68.

[54] Fabian Immler. "Formally Verified Computation of Enclosures of Solutions of Ordinary Differential Equations". In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 8430. Lecture Notes in Computer Science. Springer, 2014, pp. 113–127. DOI: 10.1007/978-3-319-06200-6_9. URL: https://doi.org/10.1007/978-3-319-06200-6%5C_9.

[55] Daniela Kaufmann, Mathias Fleury, and Armin Biere. "The Proof Checkers ck and Pastèque for the Practical Algebraic Calculus". In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 264–269. DOI: 10.34727/2020/isbn.978-3-85448-042-6_34. URL: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6%5C_34.

[56] Gerwin Klein. "Proof Engineering Considered Essential". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 16–21. DOI: 10.1007/978-3-319-06410-9_2. URL: https://doi.org/10.1007/978-3-319-06410-9%5C_2.

[57] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems* 32.1 (2014), 2:1–2:70. DOI: 10.1145/2560537. URL: https://doi.org/10.1145/2560537.

[58] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an OS kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[59] Angeliki Koutsoukou-Argyraki. "Formalising Mathematics–in Praxis; A Mathematician's First Experiences with Isabelle/HOL and the Why and How of Getting Started". In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 123.1 (2021), pp. 3–26. DOI: 10.1365/s13291-020-00221-1. URL: https://doi.org/10.1365/s13291-020-00221-1.

[60] Niklas Krafczyk and Jan Peleska. "Exhaustive Property Oriented Model-Based Testing with Symbolic Finite State Machines". In: *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*. Ed. by Radu Calinescu and Corina S. Pasareanu. Vol. 13085. Lecture Notes in Computer

Science. Springer, 2021, pp. 84–102. DOI: 10.1007/978-3-030-92124-8_5. URL: https://doi.org/10.1007/978-3-030-92124-8%5C_5.

[61] Alexander Krauss. *Defining recursive functions in Isabelle/HOL*. 2021. URL: https://isabelle.in.tum.de/dist/doc/functions.pdf (visited on 2022-04-05).

[62] Natalia Kushik, Nina Yevtushenko, and Jorge López. "Testing Against Non-deterministic FSMs: A Probabilistic Approach for Test Suite Minimization". In: *Testing Software and Systems*. Ed. by David Clark, Hector Menendez, and Ana Rosa Cavalli. Cham: Springer International Publishing, 2022, pp. 55–61. ISBN: 978-3-031-04673-5.

[63] Peter Lammich. "Automatic Data Refinement". In: *Archive of Formal Proofs* (2013-10). https://isa-afp.org/entries/Automatic_Refinement.html, Formal proof development. ISSN: 2150-914x.

[64] Peter Lammich. "Automatic Data Refinement". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 84–99. DOI: 10.1007/978-3-642-39634-2_9. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_9.

[65] Peter Lammich. "Efficient Verified (UN)SAT Certificate Checking". In: vol. 64. 3. 2020, pp. 513–532. DOI: 10.1007/s10817-019-09525-z. URL: https://doi.org/10.1007/s10817-019-09525-z.

[66] Peter Lammich. "The Imperative Refinement Framework". In: *Arch. Formal Proofs* 2016 (2016). URL: https://www.isa-afp.org/entries/Refine%5C_Imperative%5C_HOL.shtml.

[67] Peter Lammich and Andreas Lochbihler. "Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches". In: *Journal of Automated Reasoning* 63.1 (2019), pp. 53–94. DOI: 10.1007/s10817-018-9461-9. URL: https://doi.org/10.1007/s10817-018-9461-9.

[68] Pierre Letouzey. "Extraction in Coq: An Overview". In: *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Vol. 5028. Lecture Notes in Computer Science. Springer, 2008, pp. 359–369. DOI: 10.1007/978-3-540-69407-6_39. URL: https://doi.org/10.1007/978-3-540-69407-6%5C_39.

[69] Andreas Lochbihler. "Light-weight Containers". In: *Archive of Formal Proofs* (2013-04). https://isa-afp.org/entries/Containers.html, Formal proof development. ISSN: 2150-914x.

[70] Andreas Lochbihler. "Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 116–132. DOI: 10.1007/978-3-642-39634-2_11. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_11.

[71] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized WP-Method". In: *IEEE Trans. Software Eng.* 20.2 (1994), pp. 149–162. DOI: 10.1109/32.265636. URL: https://doi.org/10.1109/32.265636.

[72] Gang Luo, Alexandre Petrenko, and Gregor von Bochmann. "Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines". In: *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems*. Ed. by Tadanori Mizuno, Teruo Higashino, and Norio Shiratori. Boston, MA: Springer US, 1995, pp. 95–110. ISBN: 978-0-387-34883-4. DOI: 10.1007/978-0-387-34883-4_6. URL: https://doi.org/10.1007/978-0-387-34883-4_6.

[73] George H. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[74] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 1-55860-320-4.

[75] Tobias Nipkow. "Order-Sorted Polymorphism in Isabelle". In: *Papers Presented at the Second Annual Workshop on Logical Environments*. Edinburgh, Scotland: Cambridge University Press, 1993, pp. 164–188. ISBN: 0521433126.

[76] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*. 2021. URL: https://isabelle.in.tum.de/dist/doc/prog-prove.pdf (visited on 2022-04-05).

[77] Lawrence C. Paulson and Jasmin Christian Blanchette. "Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers". In: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, 2010, pp. 1–11. DOI: 10.29007/36dt. URL: https://doi.org/10.29007/36dt.

[78] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. "From LCF to Isabelle/HOL". In: *Formal Aspects of Computing* 31.6 (2019), pp. 675–698. DOI: 10.1007/s00165-019-00492-1. URL: https://doi.org/10.1007/s00165-019-00492-1.

[79] Lawrence C. Paulson and Kong Woei Susanto. "Source-Level Proof Reconstruction for Interactive Theorem Proving". In: *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Lecture Notes in Computer Science. Springer, 2007, pp. 232–245. DOI: 10.1007/978-3-540-74591-4_18. URL: https://doi.org/10.1007/978-3-540-74591-4%5C_18.

[80] Jan Peleska. "Model-based avionic systems testing for the airbus family". In: *23rd IEEE European Test Symposium, ETS 2018, Bremen, Germany, May 28 - June 1, 2018*. IEEE, 2018, pp. 1–10. ISBN: 978-1-5386-3728-9. DOI: 10.1109/ETS.2018.8400703. URL: https://doi.org/10.1109/ETS.2018.8400703.

[81] Jan Peleska, Jörg Brauer, and Wen-ling Huang. "Model-Based Testing for Avionic Systems Proven Benefits and Further Challenges". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11247. Lecture Notes in Computer Science. Springer, 2018, pp. 82–103. ISBN: 978-3-030-03426-9. DOI: 10.1007/978-3-030-03427-6_11. URL: https://doi.org/10.1007/978-3-030-03427-6%5C_11.

[82] Jan Peleska and Wen-ling Huang. *Test Automation - Foundations and Applications of Model-based Testing*. University of Bremen, 2021. URL: http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf.

[83] Jan Peleska, Wen-ling Huang, and Felix Hübner. "A Novel Approach to HW/SW Integration Testing of Route-Based Interlocking System Controllers". In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*. Ed. by Thierry Lecomte, Ralf Pinger, and Alexander B. Romanovsky. Vol. 9707. Lecture Notes in Computer Science. Springer, 2016, pp. 32–49. ISBN: 978-3-319-33950-4. DOI: 10.1007/978-3-319-33951-1_3. URL: http://dx.doi.org/10.1007/978-3-319-33951-1_3.

[84] Jan Peleska, Elena Vorobev, and Florian Lapschies. "Automated Test Case Generation with SMT-Solving and Abstract Interpretation". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 298–312. DOI: 10.1007/978-3-642-20398-5_22. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_22.

[85] Alexandre Petrenko. "Checking Experiments for Symbolic Input/Output Finite State Machines". In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 229–237. ISBN: 978-1-5090-3674-5. DOI: 10.1109/ICSTW.2016.9. URL: http://dx.doi.org/10.1109/ICSTW.2016.9.

[86] Alexandre Petrenko and Florent Avellaneda. "Learning and Adaptive Testing of Nondeterministic State Machines". In: *19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*. IEEE, 2019, pp. 362–373. DOI: 10.1109/QRS.2019.00053. URL: https://doi.org/10.1109/QRS.2019.00053.

[87] Alexandre Petrenko and Nina Yevtushenko. "Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs". In: *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*. Ed. by Burkhart Wolff and Fatiha Zaïdi. Vol. 7019. Lecture Notes in Computer Science. Springer, 2011, pp. 162–178. DOI: 10.1007/978-3-642-24580-0_12. URL: https://doi.org/10.1007/978-3-642-24580-0%5C_12.

[88] Alexandre Petrenko and Nina Yevtushenko. "Adaptive Testing of Nondeterministic Systems with FSM". In: *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*. IEEE Computer Society, 2014, pp. 224–228. ISBN: 978-1-4799-3465-2. DOI: 10.1109/HASE.2014.39. URL: http://dx.doi.org/10.1109/HASE.2014.39.

[89] Alexandre Petrenko and Nina Yevtushenko. "Conformance Tests as Checking Experiments for Partial Nondeterministic FSM". In: *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*. Ed. by Wolfgang Grieskamp and Carsten Weise. Vol. 3997. Lecture Notes in Computer Science. Springer, 2005, pp. 118–133. DOI: 10.1007/11759744_9. URL: https://doi.org/10.1007/11759744%5C_9.

[90] Alexandre Petrenko and Nina Yevtushenko. "Test Suite Generation from a FSM with a Given Type of Implementation Errors". In: *Protocol Specification, Testing and Verification XII, Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification, Lake Buena Vista, Florida, USA, 22-25 June 1992*. Ed. by Richard J. Linn Jr. and M. Ümit Uyar. Vol. C-8. IFIP Transactions. North-Holland, 1992, pp. 229–243.

[91] Alexandre Petrenko, Nina Yevtushenko, and Gregor von Bochmann. "Testing deterministic implementations from nondeterministic FSM specifications". In: *Testing of Communicating Systems: IFIP TC6 9th International Workshop on Testing of Communicating*

Systems Darmstadt, Germany 9–11 September 1996. Ed. by Bernd Baumgarten, Heinz-Jürgen Burkhardt, and Alfred Giessler. Boston, MA: Springer US, 1996, pp. 125–140. ISBN: 978-0-387-35062-2. DOI: 10.1007/978-0-387-35062-2_10. URL: https://doi.org/10.1007/978-0-387-35062-2_10.

[92] Arineiza C. Pinheiro, Adenilso Simão, and Ana Maria Ambrosio. "FSM-Based Test Case Generation Methods Applied to test the Communication Software on board the ITASAT University Satellite: a Case Study". In: *Journal of Aerospace Technology and Management* 6.4 (2014-11), pp. 447–461. ISSN: 2175-9146. DOI: 10.5028/jatm.v6i4.369. URL: https://doi.org/10.5028/jatm.v6i4.369.

[93] Mariana Soller Ramada, Telma Woerle de Lima, Anderson da Silva Soares, and Adenilso Simão. "Generating Reduced Tests for FSMs using a Search-Based Testing Approach". In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, pp. 400–407. DOI: 10.1109/ICTAI.2019.00063. URL: https://doi.org/10.1109/ICTAI.2019.00063.

[94] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends in Programming Languages* 5.2-3 (2019), pp. 102–281. DOI: 10.1561/2500000045. URL: https://doi.org/10.1561/2500000045.

[95] Robert Sachtleben. "An Executable Mechanised Formalisation of an Adaptive State Counting Algorithm". In: *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*. Ed. by Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak. Vol. 12543. Lecture Notes in Computer Science. Springer, 2020, pp. 236–254. DOI: 10.1007/978-3-030-64881-7_15. URL: https://doi.org/10.1007/978-3-030-64881-7%5C_15.

[96] Robert Sachtleben, Robert M. Hierons, Wen-ling Huang, and Jan Peleska. "A Mechanised Proof of an Adaptive State Counting Algorithm". In: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*. Ed. by Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall. Vol. 11812. Lecture Notes in Computer Science. Springer, 2019, pp. 176–193. DOI: 10.1007/978-3-030-31280-0_11. URL: https://doi.org/10.1007/978-3-030-31280-0%5C_11.

[97] Robert Sachtleben and Jan Peleska. "Effective grey-box testing with partial FSM models". In: *Software Testing, Verification and Reliability* 32.2 (2022), e1806. DOI: https://doi.org/10.1002/stvr.1806. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1806.

URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1806.

[98] Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. "A verified prover based on ordered resolution". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 152–165. DOI: 10.1145/3293880.3294100. URL: https://doi.org/10.1145/3293880.3294100.

[99] Natalia Shabaldina and Maxim Gromov. "FSMTest-1.0: A manual for researches". In: *2015 IEEE East-West Design & Test Symposium, EWDTS 2015, Batumi, Georgia, September 26-29, 2015*. IEEE Computer Society, 2015, pp. 1–4. DOI: 10.1109/EWDTS.2015.7493141. URL: https://doi.org/10.1109/EWDTS.2015.7493141.

[100] Adenilso da Silva Simão and Alexandre Petrenko. "Checking Completeness of Tests for Finite State Machines". In: *IEEE Trans. Computers* 59.8 (2010), pp. 1023–1032. DOI: 10.1109/TC.2010.17. URL: https://doi.org/10.1109/TC.2010.17.

[101] Adenilso da Silva Simão and Alexandre Petrenko. "Fault Coverage-Driven Incremental Test Generation". In: *Comput. J.* 53.9 (2010), pp. 1508–1522. DOI: 10.1093/comjnl/bxp073. URL: https://doi.org/10.1093/comjnl/bxp073.

[102] Adenilso da Silva Simão, Alexandre Petrenko, and Nina Yevtushenko. "Generating Reduced Tests for FSMs with Extra States". In: *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*. Ed. by Manuel Núñez, Paul Baker, and Mercedes G. Merayo. Vol. 5826. Lecture Notes in Computer Science. Springer, 2009, pp. 129–145. DOI: 10.1007/978-3-642-05031-2_9. URL: https://doi.org/10.1007/978-3-642-05031-2%5C_9.

[103] Adenilso da Silva Simão, Alexandre Petrenko, and Nina Yevtushenko. "On reducing test length for FSMs with extra states". In: *Software Testing, Verification and Reliability* 22.6 (2012), pp. 435–454. DOI: 10.1002/stvr.452. URL: https://doi.org/10.1002/stvr.452.

[104] Michal Soucha. "Checking experiment design methods". MA thesis. Czech Technical University in Prague, 2015.

[105] Michal Soucha. *FSMlib*. 2018. URL: https://github.com/Soucha/FSMlib (visited on 2022-04-05).

[106] Michal Soucha. "Testing and active learning of resettable finite-state machines". PhD thesis. University of Sheffield, 2019.

[107] Michal Soucha and Kirill Bogdanov. "Observation Tree Approach: Active Learning Relying on Testing". In: *The Computer Journal* 63.9 (2020), pp. 1298–1310. DOI: 10.1093/comjnl/bxz056. URL: https://doi.org/10.1093/comjnl/bxz056.

[108] Michal Soucha and Kirill Bogdanov. "SPYH-Method: An Improvement in Testing of Finite-State Machines". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 194–203. DOI: 10.1109/ICSTW.2018.00050. URL: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00050.

[109] Mark Staples, D. Ross Jeffery, June Andronick, Toby C. Murray, Gerwin Klein, and Rafal Kolanski. "Productivity for proof engineering". In: *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*. Ed. by Maurizio Morisio, Tore Dybå, and Marco Torchiano. ACM, 2014, 15:1–15:4. DOI: 10.1145/2652524.2652551. URL: https://doi.org/10.1145/2652524.2652551.

[110] Jan Tretmans. "Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation". In: *Computer Networks and ISDN Systems* 29.1 (1996), pp. 49–79. DOI: 10 . 1016 / S0169 - 7552(96) 00017 - 7. URL: https://doi.org/10.1016/S0169-7552(96)00017-7.

[111] Jan Tretmans. "Model based testing with labelled transition systems". In: *Formal methods and testing*. Springer, 2008, pp. 1–38.

[112] Hasan Ural, Xiaolin Wu, and Fan Zhang. "On Minimizing the Lengths of Checking Sequences". In: *IEEE Transactions on Computers* 46.1 (1997), pp. 93–99. DOI: 10.1109/12.559807. URL: https://doi.org/10.1109/12.559807.

[113] Mark Utting. "The Role of Model-Based Testing". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer, 2005, pp. 510–517. DOI: 10.1007/978-3-540-69149-5_56. URL: https://doi.org/10.1007/978-3-540-69149-5%5C_56.

[114] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software testing, verification and reliability* 22.5 (2012), pp. 297–312. DOI: 10.1002/stvr.456. URL: https://doi.org/10.1002/stvr.456.

[115] M. P. Vasilevskii. "Failure diagnosis of automata". In: *Kibernetika (Transl.)* 4 (1973), pp. 98–108.

[116] Evgenii M. Vinarskii, Andrey Laputenko, and Nina Yevtushenko. "Using an SMT Solver for Checking the Completeness of FSM-Based Tests". In: *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*. Ed. by Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak. Vol. 12543. Lecture Notes in Computer Science. Springer, 2020, pp. 289–295. DOI: 10.1007/978-3-030-64881-7_18. URL: https://doi.org/10.1007/978-3-030-64881-7%5C_18.

[117] Lucas G. Wagner, Alain Mebsout, Cesare Tinelli, Darren D. Cofer, and Konrad Slind. "Qualification of a Model Checker for Avionics Software Verification". In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. Ed. by Clark W. Barrett, Misty Davies, and Temesghen Kahsai. Vol. 10227. Lecture Notes in Computer Science. 2017, pp. 404–419. DOI: 10.1007/978-3-319-57288-8_29. URL: https://doi.org/10.1007/978-3-319-57288-8%5C_29.

[118] Haitao Wang and Lihua Song. "Study of Isabelle/HOL on Formal Algorithm Analysis and Code Generation". In: *IEEE Access* 9 (2021), pp. 25002–25013. DOI: 10.1109/ACCESS.2021.3057398. URL: https://doi.org/10.1109/ACCESS.2021.3057398.

[119] Makarius Wenzel. *The isabelle/isar reference manual*. 2021. URL: https://isabelle.in.tum.de/dist/doc/isar-ref.pdf (visited on 2022-04-05).

[120] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. "The Isabelle Framework". In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 33–38. DOI: 10.1007/978-3-540-71067-7_7. URL: https://doi.org/10.1007/978-3-540-71067-7%5C_7.

[121] Markus Wenzel. "Isabelle, Isar - a versatile environment for human readable formal proof documents". PhD thesis. Technical University Munich, Germany, 2002. URL: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf.

[122] Markus Wenzel. "Type Classes and Overloading in Higher-Order Logic". In: *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*. Ed. by Elsa L. Gunter and Amy P. Felty. Vol. 1275. Lecture Notes in Computer Science. Springer, 1997, pp. 307–322. DOI: 10.1007/BFb0028402. URL: https://doi.org/10.1007/BFb0028402.

[123] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *Proceedings of the 5th ACM*

SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016. Ed. by Jeremy Avigad and Adam Chlipala. ACM, 2016, pp. 154–165. DOI: 10.1145/2854065.2854081. URL: https://doi.org/10.1145/2854065.2854081.

[124] Nina Yevtushenko, Victor V. Kuliamin, and Natalia Kushik. "Evaluating the Complexity of Deriving Adaptive Homing, Synchronizing and Distinguishing Sequences for Nondeterministic FSMs". In: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*. Ed. by Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall. Vol. 11812. Lecture Notes in Computer Science. Springer, 2019, pp. 86–103. DOI: 10.1007/978-3-030-31280-0_6. URL: https://doi.org/10.1007/978-3-030-31280-0%5C_6.

Appendices

Appendix A

Reduction Testing

Language-equivalence is only one among several conformance relations developed over finite state machines. Reduction, described in Section 3.1, constitutes another important conformance relation that has been intensely studied and for which several test strategies have been developed. For example, Petrenko et al. in [88] and Hierons in [44] employ so called *adaptive state counting strategies* for generating m-complete test suites for reduction testing. As it may be possible for the behaviour of the system under test to conflate states of the reference model while still being a reduction of it, it is more difficult to separate traces with respect to reduction than it is with respect to language-equivalence, leading to the use of *adaptive test cases* (see [32, 33, 124]).

To my best knowledge, by mechanising the completeness proof of [44], Hierons, Huang, Peleska and I present in [96] the first mechanised proof of a complete strategy for testing reduction. This effort also uncovered and clarified an ambiguity in the natural language specification of the algorithm, which could lead to the generation of test suites of insufficient strength. In contrast to the present work, this effort did not yet consider the generation of trustworthy executable code from the mechanisation. In consequence, the resulting implementations are not well suited for code generation. Therefore, in [95] I develop a mechanised formalisation of the strategy described in [88] that is suitable to automatically generate trustworthy implementations from. This mechanisation effort also constitutes the foundation of the present work, which reuses many basic properties on FSMs. However, for the purposes of implementing test strategies for language-equivalence, this reuse is limited by the fact that strategies for reduction testing share few algorithmic steps with those for language-equivalence testing. The original Isabelle/HOL files of this mechanisation effort, together with corresponding documentation, remain publicly available for download at `https://bitbucket.org/RobertSachtleben/an-executable-formalisation-of-an-adaptive-state-counting`. Including minor modifications and several updated names, these files are also part of the Isabelle/HOL files publicly available for the present work (see Section 1.4).

The formalisation developed in [95] already employs the concept of inter-

face lemmata (see Subsection 6.2.2) by introducing a sufficient condition for m-completeness (w.r.t. reduction) over test suites. The completeness proof of the formalised strategy is then split into a first proof that any test suite satisfying this condition is m-complete, and a second proof that the strategy described in [88] satisfies the condition. As the latter in turn relies on certain properties of functions employed within the strategy, the formalisation may easily be rewritten as a framework in the sense discussed in Section 6.2. Similar to the option of implementing the *safety-complete* H-Method described in [47] via the PAIR-FRAMEWORK as discussed in 6.5, a framework for state counting strategies might also be of use in creating mechanised formalisations of strategies closely related to state counting. These include the strategy for testing against the *strong reduction* conformance relation that Peleska and I have proposed in [97], as well as strategies for testing the *quasi-reduction* conformance relation such as that presented by Petrenko et al. in [89].

Appendix B

IO-Based Test Cases

Test strategies for language-equivalence testing on finite state machines classically employ input sequences as test cases, where FSM I passes input sequence $\bar{x}$ w.r.t. reference model M, if I and M exhibit the same responses to $\bar{x}$ and all of its prefixes. In the present work, I instead employ IO-traces as test cases, where I passes α w.r.t. M if for each prefix of α it holds that this prefix is contained in either both $\mathcal{L}(M)$ and $\mathcal{L}(I)$ or in neither language.

This choice has been made for two reasons. First, in the context of possibly nondeterministic OFSMs, an OFSM may exhibit more than one response to some input sequence, but not all of these are necessarily of interest in testing. For example, states q_1 and q_2 of M_H depicted in Fig. B.1 are distinguished by IO-trace $ab/10 \in \mathcal{L}_{M_H}(q_2) \setminus \mathcal{L}_{M_H}(q_1)$, whereas $ab/00$ is exhibited by both states. Thus, in order to check whether $ab/10$ distinguishes two states of an implementation, it is not necessary to apply further inputs after observing response 0 to first input a of $ab/10$. In contrast, if test cases are represented as input sequences, then input portion ab of $ab/10$ would have to be applied fully, regardless of the output observed to first input a. That is, (sets of) IO-traces may be viewed as *adaptive test cases* (see [44, 88]), where the responses observed to previously applied inputs influence subsequent inputs.

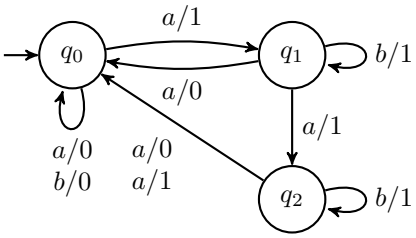


Figure B.1: FSM M_H.

Second, any IO-trace applied to some OFSM reaches at most one state, whereas an input sequence may reach zero, one, or many states. Thus, the use

of IO-traces reduces the need to quantify over multiple distinct states reached by a test case. This, in turn, may also lead to smaller test suites for test strategies that extend test cases α based on the states reached by α in the reference model.

As an example of this potential reduction, consider again FSM M_H depicted in Fig. B.1, with $\Sigma_I = \{a, b\}$ and $\Sigma_O = \{0, 1\}$. Set $V = \{\epsilon, a/1, aa/11\}$ is a state cover of M_H. Furthermore consider the following sets:

$$A := V \times V$$
$$B := V \times (V.(\Sigma_I \times \Sigma_O)) = V \times (V.\{a/0, a/1, b/0, b/1\})$$

Let $A|_{\Sigma_I}$ and $B|_{\Sigma_I}$ denote the input projections of A and B, respectively. A 3-complete test suite using the H-Method, originally presented in [27] and described in Subsection 5.1.5 of the present work, is created by initialising the test suite as $V|_{\Sigma_I} \cup V|_{\Sigma_I}.\Sigma_I$ and then choosing for each $(\alpha, \beta) \in A|_{\Sigma_I} \cup B|_{\Sigma_I}$ where α and β reach distinct states q_α, q_β in M_H some input sequence γ that distinguishes them, adding $\alpha.\gamma$ and $\beta.\gamma$ to the test suite. Note that $A|_{\Sigma_I} \cup B|_{\Sigma_I} = \{\epsilon, a, aa\} \times \{\epsilon, a, b, aa, ab, aaa, aab\}$ and that state q_0 can be distinguished from all other states via $b/0$, while q_1 and q_2 can be distinguished by $ab/10$. Table B.1 shows one possibility of appending distinguishing γ as required. Let TS_1 denote the resulting test suite, omitting all non-maximal test cases:

$$TS_1 := \{bb, abb, aaab, aabb, abab, aaaab, aabab\}$$

As discussed in 5.1.5, the H-Method can easily be modified to work on IO-traces instead of input sequences. In this strategy, a 3-complete test suite for M_H is initialised as $V \cup V.(\Sigma_I \times \Sigma_O)$ and for each $(\alpha, \beta) \in A \cup B$, if α, β are contained in $\mathcal{L}(M_H)$ and reach distinct states in M_H, then some trace γ that distinguishes those states is chosen and $\alpha.\gamma, \beta.\gamma$ are added to the test suite. Note that $A \cup B = \{\epsilon, a/1, aa/11\} \cup (\{\epsilon, a/1, aa/11\} \times \{a/0, a/1, b/0, b/1\})$. One possibility of choosing such γ is shown in Table B.2. Let $TS_2|_{\Sigma_I}$ denote the input projection of the resulting test suite, restricted to maximal test cases:

$$TS_2|_{\Sigma_I} := \{bb, abb, aaab, aabb, abab, aabab\}$$

Thus, $TS_2|_{\Sigma_I}$ is smaller than TS_1, as the former does not contain $aaaab$, which is only added to TS_1 while handling (a, aaa) or (aa, aaa), which necessitates distinguishing q_1 from both $q_0, q_2 \in M_H$-after-aaa. In TS_2, this aaa is only considered as input portion of $aaa/110$ and $aaa/111$, both of which reach q_0.

The two test suites TS_1 and TS_2 also serve to illustrate the first benefit described above of IO-traces as test suites, as TS_2 contains only one test case beginning with $a/0$, namely $ab/00$. Thus, in applying TS_2 to some system under test, the application of test cases such as $aaab/1110$ may be aborted if output 0 is observed in response to the second input a. In contrast, TS_1 does not store such information and has to apply $aaab$ fully. Note, however, that the use of IO-traces as test cases can substantially increase the amount of space required to store test suites, as there exist $|\Sigma_I \times \Sigma_O|^l$ traces of length l for input alphabet Σ_I and output alphabet Σ_O, whereas there are only $|\Sigma_I|^l$ input sequence of length l over Σ_I.

Table B.1: Choices of distinguishing sequences to add for M_H and $V|_{\Sigma_I}$ using the H-Method with input sequences. Columns q_α and q_β list all states of M_H reached by input sequence α and β, respectively. Dashes in the column for distinguishing sequences (dist. seq.) indicate that the traces do not need to be separated as they converge in M_H.

α	β	q_α	q_β	dist. seq.	added test cases
ϵ	ϵ	$\{q_0\}$	$\{q_0\}$	-	
ϵ	a	$\{q_0\}$	$\{q_0, q_1\}$	b	b, ab
ϵ	b	$\{q_0\}$	$\{q_1\}$	b	b, bb
ϵ	aa	$\{q_0\}$	$\{q_0, q_1, q_2\}$	b	b, aab
ϵ	ab	$\{q_0\}$	$\{q_0, q_1\}$	b	b, abb
ϵ	aaa	$\{q_0\}$	$\{q_0, q_1, q_2\}$	b	$b, aaab$
ϵ	aaa	$\{q_0\}$	$\{q_0, q_1, q_2\}$	b	$b, aabb$
a	ϵ	$\{q_0, q_1\}$	$\{q_0\}$	b	ab, b
a	a	$\{q_0, q_1\}$	$\{q_0, q_1\}$	b	ab
a	b	$\{q_0, q_1\}$	$\{q_1\}$	b	ab, bb
a	aa	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	b, ab	$ab, aab, aaab$
a	ab	$\{q_0, q_1\}$	$\{q_0, q_1\}$	b	ab, abb
a	aaa	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	b, ab	$ab, aab, aaab, aaaab$
a	aab	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	b, ab	$ab, aab, aabb, aabab$
aa	ϵ	$\{q_0, q_1, q_2\}$	$\{q_0\}$	b	aab, b
aa	a	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	b, ab	$aab, aaab, ab$
aa	b	$\{q_0, q_1, q_2\}$	$\{q_0\}$	b	aab, bb
aa	aa	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	b, ab	$aab, aaab$
aa	ab	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	b, ab	$aab, aaab, abb, abab$
aa	aaa	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	b, ab	$aab, aaab, aaaab$
aa	aab	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	b, ab	$aab, aaab, aabb, aabab$

Table B.2: Choices of distinguishing traces to add for M_H and V using the H-Method with IO-traces. Columns q_α and q_β respectively list states q_0-after-α and q_0-after-β, where a dash indicates $\beta \notin \mathcal{L}(M_H)$ and hence that the respective α, β need not be separated. Similarly, dashes in the column for distinguishing traces (dist. tr.) indicate that the traces do not need to be separated as they converge in M_H.

α	β	q_α	q_β	dist. tr.	test cases
ϵ	ϵ	q_0	q_0	-	
ϵ	$a/0$	q_0	q_0	-	
ϵ	$a/1$	q_0	q_1	$b/0$	$b/0, ab/10$
ϵ	$b/0$	q_0	q_0	-	
ϵ	$b/1$	q_0	-		
ϵ	$aa/10$	q_0	q_0	-	
ϵ	$aa/11$	q_0	q_2	$b/0$	$b/0, aab/110$
ϵ	$ab/10$	q_0	-		
ϵ	$ab/11$	q_0	q_1	$b/0$	$b/0, abb/110$
ϵ	$aaa/110$	q_0	q_0	-	
ϵ	$aaa/111$	q_0	q_0	-	
ϵ	$aab/110$	q_0	-		
ϵ	$aab/111$	q_0	q_2	$b/0$	$b/0, aabb/1110$
$a/1$	$a/0$	q_1	q_0	$b/0$	$ab/10, ab/00$
$a/1$	$a/1$	q_1	q_1	-	
$a/1$	$b/0$	q_1	q_0	$b/0$	$ab/10, bb/00$
$a/1$	$b/1$	q_1	-		
$a/1$	$aa/10$	q_1	q_0	$b/0$	$ab/10, aab/100$
$a/1$	$aa/11$	q_1	q_2	$ab/10$	$aab/110, aaab/1110$
$a/1$	$ab/10$	q_1	-		
$a/1$	$ab/11$	q_1	q_1	-	
$a/1$	$aaa/110$	q_1	q_0	$b/0$	$ab/10, aaab/1100$
$a/1$	$aaa/111$	q_1	q_0	$b/0$	$ab/10, aaab/1110$
$a/1$	$aab/110$	q_1	-		
$a/1$	$aab/111$	q_1	q_2	$ab/10$	$aab/110$, $aabab/11110$
$aa/11$	$a/0$	q_2	q_0	$b/0$	$aab/110, ab/00$
$aa/11$	$b/0$	q_2	q_0	$b/0$	$aab/110, bb/00$
$aa/11$	$b/1$	q_2	-		
$aa/11$	$aa/10$	q_2	q_0	$b/0$	$aab/110, aab/100$
$aa/11$	$aa/11$	q_2	q_2	-	
$aa/11$	$ab/10$	q_2	-		
$aa/11$	$ab/11$	q_2	q_1	$ab/10$	$aaab/1110$, $abab/1110$
$aa/11$	$aaa/110$	q_2	q_0	$b/0$	$aab/110, aaab/1100$
$aa/11$	$aaa/111$	q_2	q_0	$b/0$	$aab/110, aaab/1110$
$aa/11$	$aab/110$	q_2	-		
$aa/11$	$aab/111$	q_2	q_2	-	

Appendix C

Prefix Trees

Defined in theory file `Prefix_Tree.thy`, data type `prefix_tree` describes a tree with unlabelled nodes and labelled edges, stored as a map assigning to each edge-label the corresponding sub-tree, if such a sub-tree exists. For edge-labels of arbitrary type `'a`, this is implemented in Isabelle/HOL in the following data type, using constructor `PT` whose single parameter is a map[1] from `'a` to prefix trees using the same type of edge-label:

```
datatype 'a prefix_tree = PT "'a ⇀ 'a prefix_tree"
```

In this work, prefix trees are employed to store prefix-closed sets of traces, where a trace is contained in a prefix tree if and only if it constitutes the sequence of edge-labels of a path from the root of the tree. Table C.1 lists the most important operations on prefix trees as provided by `Prefix_Tree.thy`. As they are defined using `datatype`, prefix trees are by construction of finite height (see [10]). Thus, function `finite_tree` described in Table C.1 effectively only checks whether the map of each node has a finite domain. Note that a prefix tree containing no edges and only the root node represents the set $\{\epsilon\}$ and that hence data type `prefix_tree` is unable to represent the empty set.

As discussed in Section 12.4, in order to use efficient representations for maps via the *Containers* framework, it is necessary to refine the usage of `map` to instead employ type `mapping`. For prefix trees, this is implemented via a new constructor `MPT` using a `mapping`, from which the `map` required in `PT` is obtained by partial application of a lookup operation on the `mapping`:

```
definition MPT :: "('a,'a prefix_tree) mapping ⇒ 'a prefix_tree" where
  "MPT m = PT (Mapping.lookup m)"
```

[1] Notation `('a ⇀ 'b)` is an alternative representation of type `(('a,'b) map)` for maps with keys of type `'a` and values of type `'b`.

Table C.1: Operations provided by `Prefix_Tree.thy`. The second column provides the semantics of each function, interpreting prefix trees as prefix-closed sets.

Function	Set-interpretation
`after T` α	$\{\epsilon\} \cup \{\beta \mid \alpha.\beta \in$ `T`$\}$
`combine T1 T2`	`T1` $\cup$ `T2`
`combine_after T1` α `T2`	`T1` $\cup$ *pref*$(\{\alpha\}.$`T2`$)$
`difference_list T1 T2`	sorted list of all traces in `T1` $\setminus$ `T2`
`empty`	$\{\epsilon\}$
`finite_tree T`	`True` iff `T` is finite
`from_list xs`	*pref*$(\{\alpha \mid \alpha \in$ `(List.set xs)`$\})$
`height T`	length of a maximum length trace in `T`
`insert T` α	`T` $\cup$ *pref*$(\{\alpha\})$
`isin T` α	`True` iff $\alpha \in$ `T`
`is_leaf T`	`True` iff `T` $\equiv$ `empty`
`is_maximal_in T` α	`True` iff α is a maximal length trace in `T`
`maximum_prefix T` α	maximum length trace α' in *pref*$(\alpha) \cap$ `T`
`set T`	`T` as value of type `set`
`sorted_list_of_maximal_sequences_in_tree T`	sorted list of all maximum length traces in `T`
`sorted_list_of_sequences_in_tree T`	sorted list of all traces in `T`

Appendix D

Overview of Defined Functions

In the following sections, I provide an overview over the functions defined in the theory files described in Part III. This listing omits functions defined only for the purpose of code refinement, as well as most functions used only in the implementation of test strategies for reduction testing. Operations on prefix trees are detailed in Appendix C.

D.1 FSM_Impl.thy

`add_input M x`

Creates an `fsm_impl` from `M` by adding input `x` to the inputs of `M`.

`add_output M y`

Creates an `fsm_impl` from `M` by adding output `y` to the outputs of `M`.

`add_state M q`

Creates an `fsm_impl` from `M` by adding state `q` to the states of `M`.

`add_transition M t`

Creates an `fsm_impl` from `M` and transition `t` by adding `t` to the transitions of `M`. If any component of `t` is not already contained in the inputs, outputs, or states of `M`, then `M` is returned.

`add_transition_with_components M t`

Creates an `fsm_impl` from `M` and transition `t` by adding `t` and its components to `M`.

`add_transitions M ts`

Creates an `fsm_impl` from `M` and list of transitions `ts` by adding all transitions in `ts` to `M`. If any transition in `ts` uses a component not already contained in the inputs, outputs, or states of `M`, then `M` is returned.

`create_unconnected_fsm_from_fsets q ns ins outs`

Creates an `fsm_impl` with initial state `q`, state set ($\{$`q`$\} \cup$ `fset ns`), input set `fset ins`, output set `fset outs`, and no transitions, where `fset` is a predefined function that obtains an `'a set` from a given `'a fset`.

`filter_states M P`

Creates an `fsm_impl` from `M` by removing all states `q` such that `P q` does not hold. All transitions containing such `q` are removed. If `P` does not hold for the initial state of `M`, then `M` is returned.

`filter_transitions M P`

Creates an `fsm_impl` from `M` by removing all transitions `t` that do not satisfy `P`.

`from_FSMI M q`

Creates an `fsm_impl` from `M` by replacing its initial state with `q` if `q` is already a state of `M`. Otherwise returns `M`.

`fsm_impl_from_list q ts`

Creates an `fsm_impl` from state `q` and transition list `ts`. If `ts` is empty, then the initial and only state is `q` and inputs, outputs, and transitions are empty sets. Otherwise, the initial state is the source of the first transition in `ts` and states, inputs, outputs are obtained by collecting the respective components in `ts`, while the transition set is the set of all elements in `ts`.

`h M (q,x)`

Returns the set of all `(y,q')` such that `(q,x,y,q')` is a transition of `M`.

`h_obs M q x y`

Returns `(Some q')` if `q'` is the only state such that `(q,x,y,q')` is a transition of `M`. If no such `q'` exists, `None` is returned.

`initial M`

Returns the initial state of `M`.

`inputs M`

Returns the input alphabet of `M`.

`outputs M`

Returns the output alphabet of `M`.

`rename_states M f`

Creates an `fsm_impl` from `M` by renaming to `(f q)` each state `q` of `M`. The renaming is also applied to the transitions and the initial state of `M`.

`states M`

Returns the state set of `M`.

`t_input t`

Returns the input of transition `t`

`t_output t`

Returns the output of transition `t`

`t_source t`

Returns the source of transition `t`

`t_target t`

Returns the target of transition `t`

`transitions M`

Returns the transition set of `M`.

D.2 FSM.thy

`acyclic M`

Checks whether `M` is acyclic. That is, it returns `True` if and only if the language of `M` is finite.

`acyclic_paths_up_to_length M q k`

Returns the set of all paths in `M` from state `q` of length up to `k` that do not contain cycles.

`add_input M x`

Lifted version of `add_input` from Appendix D.1.

`add_output M y`

Lifted version of `add_output` from Appendix D.1.

`add_state M q`

Lifted version of `add_state` from Appendix D.1.

`add_transition M t`

Lifted version of `add_transition` from Appendix D.1.

`add_transition_with_components M t`

Lifted version function of `add_transition_with_components` from Appendix D.1.

`add_transitions M ts`

Lifted version of `add_transitions` from Appendix D.1.

`after M q io`

If there exists a unique path in `M` from state `q` with IO-projection `io`, then the target of this path is returned (which is `q` if the path is empty). Otherwise, returns `undefined`.

`after_initial M io`

Applies `after` to the initial state of `M` using `io`.

`completely_specified M`

Checks whether `M` is completely specified.

`completely_specified_state M q`

Checks whether there exists in `M` a transition from state `q` for every input if `M`.

`create_unconnected_fsm_from_fsets q ns ins outs`

Lifted version of `create_unconnected_fsm_from_fsets` introduced in Appendix D.1.

`defined_inputs M q`

Returns the set of all inputs `x` such that there exists a transition in `M` from state `q` with input `x`.

`deterministic M`

Checks whether `M` is deterministic.

`distinguishes M q1 q2 io`

Checks whether trace `io` distinguishes states `q1` and `q2` in `M`.

`does_distinguish M q1 q2 io`

Checks whether there exists in `M` a unique path from exactly one element of `{q1,q2}` whose IO-projection is `io`. In observable FSMs, this is equivalent to checking whether trace `io` distinguishes states `q1` and `q2` in `M`.

`equal_fsm M1 M2`

Checks whether FSMs `M1` and `M2` use equivalent initial states, state sets, input sets, output sets, and transition sets.

`filter_states M P`

Lifted version of `filter_states` from Appendix D.1.

`filter_transitions M P`

Lifted version of `filter_transitions` from Appendix D.1.

`finputs M`

Returns a finite set (`fset`) containing the inputs of `M`.

`foutputs M`

Returns a finite set (`fset`) containing the outputs of `M`.

`from_FSM M q`

Lifted version of `from_FSMI` from Appendix D.1.

`fsm_from_list q ts`

Lifted version of `fsm_impl_from_list` from Appendix D.1.

`fstates M`

Returns a finite set (`fset`) containing the states of `M`.

`ftransitions M`

Returns a finite set (`fset`) containing the transitions of `M`.

`h M (q,x)`

Lifted version of `h` from Appendix D.1.

`h_from M q`

Returns the set of all `(x,y,q')` such that `(q,x,y,q')` is a transition of `M`.

`h_obs M q x y`

Lifted version of `h_obs` from Appendix D.1.

`h_out M q x`

Returns the set of all `y` such that there exists some state `q'` such that `(q,x,y,q')` is a transition of `M`.

`index_states M`

Renames the states of `M` to natural numbers between 0 and `(|M| - 1)`.

`index_states_integer M`

Renames the states of `M` to integers between 0 and `(|M| - 1)`.

`initial M`

Lifted version of `initial` from Appendix D.1.

`input_portion io`

Returns the input portion of IO-trace `io`.

`inputs M`

Lifted version of `inputs` from Appendix D.1.

`inputs_as_list M`

Returns a list containing the inputs of `M` without duplicates.

`io_targets M io q`

Returns the set of all states reachable in `M` from state `q` by valid paths whose IO-projection is `io`.

`is_in_language M q io`

Checks whether there exists a unique path in `M` from state `q` with IO-projection `io`. In observable FSMs, this is equivalent to checking whether `io` is in the language of `q` in `M`.

`L M`

Returns the language of `M`.

`language_for_input M q xs`

Returns the set of all traces `io` in the language of state `q` for `M` such that the input projection of `io` is `xs` and there exists a unique path from `q` in `M` whose IO-projection is `io`. In observable FSMs this, is equivalent to returning the set of all traces in the language of state `q` for `M` whose input projection is `xs`. If `xs` is empty, then the empty trace is returned, regardless of whether `q` is a state of `M`.

`language_state_for_input M q xs`

Returns the set of all traces in the language of state `q` for `M` whose input projection is `xs`.

`LS M q`

Returns the language of state `q` in `M`.

`maximal_prefix_in_language M q io`

Returns the maximal prefix `io'` of IO-trace `io` such that there exists a unique path in `M` from state `q` whose IO-projection is `io'`. In observable FSMs, this is equivalent to returning the maximal prefix of IO-trace `io` that is in the language of state `q` in `M`.

`minimal M`

Checks whether `M` is minimal.

`minimally_distinguishes M q1 q2 io`

Checks whether trace `io` distinguishes states `q1` and `q2` in `M` and no trace shorter than `io` distinguishes these states.

`observable M`

Checks whether `M` is observable.

`outputs M`

Lifted version of `outputs` from Appendix D.1.

`outputs_as_list M`

Returns a list containing the outputs of `M` without duplicates.

`paths_for_input M q xs`

Returns the set of all valid paths in `M` from state `q` whose IO-projection has input portion `io`.

`paths_for_io M q io`

Returns the set of all valid paths in `M` from state `q` whose IO-projection is `io`.

`paths_of_length M q k`

Returns the set of all valid paths in `M` from state `q` of length `k`. If `k` is 0, then the singleton `{[]}` is returned, regardless of whether `q` is a state of `M`.

`paths_up_to_length M q k`

Returns the set of all valid paths in `M` from state `q` of length up to `k`. If `k` is 0, then the singleton `{[]}` is returned, regardless of whether `q` is a state of `M`.

`paths_up_to_length_or_condition_with_witness M P k q`

Returns the set of all `(p,w)` such that `p` is a valid path in `M` from state `q`, `p` is of length at most `k`, `P p = Some w`, and for all proper prefixes of `p` it holds that `P p = None`.

`p_io p`

Returns the IO-projection of path `p`.

`reachable M q`

Checks whether state `q` is a reachable state of `M`.

`reachable_k M q n`

Checks whether there exists a path in `M` from the initial state of `M` to state `q` that is of length at most `n`.

`reachable_states M`

Returns the set of all reachable states of `M`.

`reachable_states_as_list M`

Returns a list containing the reachable states of `M` without duplicates.

`rename_states M f`

Lifted version of `rename_states` from Appendix D.1.

`restrict_to_reachable_states M`

Creates an `fsm` from `M` by removing all non-reachable states and any transitions that contain non-reachable states.

`size M`

Returns the number of states of `M`.

`size_r M`

Returns the number of reachable states of `M`.

`states M`

Lifted version of `states` from Appendix D.1.

`states_as_list M`

Returns a list containing the states of `M` without duplicates.

`target q p`

Returns the state reached by path `p` from state `q`. That is, returns the target of the last transition in `p`, if some exists. If `p` is empty, then `q` is returned.

`transitions M`

Lifted version of `transitions` from Appendix D.1.

`transitions_as_list M`

Returns a list containing the transitions of `M` without duplicates.

`transitions_from M q`

Returns the set of all transitions in `M` from state `q`.

`visited_states q p`

Returns a list beginning with state `q` and continuing with the targets of the transitions in path `p` in the same order as they appear in `p`, including duplicates.

`well_formed_fsm M`

Checks for `fsm_impl M` whether its initial state is contained in its state set, whether the state, input, output, and transition sets are finite and whether the components of every transition in `M` are contained in its state, input, and output sets.

D.3 State_Cover.thy

`covered_transitions M V io`

Returns the set of all transitions `(q,x,y,q')` in the unique path in `M` from the initial state of `M` whose IO-trace is `io` such that `(V q)@[(x,y)]` is the same trace as `(V q')`, where predefined function `@` appends lists. If no such unique path exists, then the result is `undefined`. In observable FSMs, this function returns all transitions considered *covered* (w.r.t. the SPY and SPYH-Methods) for state cover assignment `V` along trace `io`.

`get_state_cover M`

Returns a state cover assignment for `M`, to be computed using a breadth-first-search.

`is_minimal_state_cover M SC`

Checks whether `SC` is a state cover of `M` such that no proper subset of `SC` is a state cover of `M`.

`is_state_cover M SC`

Checks whether `SC` contains the empty trace and for each reachable state `q` of `M` a trace reaching `q`.

`is_state_cover_assignment M V`

Checks whether `V` assigns the empty trace to the initial state of `M` and also assigns to each reachable state `q` of `M` a trace reaching `q`.

`reachable_nodes_up_to_depth M nx done V k`

Implementation of Algorithm 39.

D.4 Observability.thy

`make_observable_transitions ts_base nx dones ts_obs`

Implementation of Algorithm 40.

`make_observable M`

Creates an observable `fsm M'` that is language-equivalent to `M`. The states of `M'` are finite sets (`fset`) of states of `M`. The initial state of `M'` is the `fset` containing only the initial state of `M`. `M'` retains the inputs and outputs of `M`.

D.5 Minimisation.thy

`equivalence_relation_on_states M f`

Checks whether $\{(q1, q2) \mid q1 \in$ `states M` $\wedge\, q2 \in$ `f` $q1\}$ is an equivalence relation over the states of `M` and whether for all states `q` of `M` the set `(f q)` is a subset of the states of `M`.

`minimise M`

Returns an `fsm M'` that is language-equivalent to `M` if `M` is observable. Otherwise the result is not specified.

`ofsm_table M f k q`

Returns the set of all states in the same class as `q` in the `k`-th OFSM-Table, using `f` as the initial assignment to classes. If `M` is not observable, then the result is not specified.

`ofsm_table_fix M f k`

Returns the `n`-th OFSM-Table for `M` and `f` for the smallest `n` $\geq$ `k` such that the OFSM-Tables for `M` do not change after the `n`-th table. Restricts `f` to produce only subsets of the states of `M`. If `M` is not observable, then the result is not specified.

D.6 Distinguishability.thy

`assemble_distinguishing_sequence_from_ofsm_table M q1 q2 k`

Returns an IO-trace `io` of length at most `k` that distinguishes `q1`, `q2` in `M`. If `M` is not observable or no such `io` exists, then the result is not specified.

`find_first_distinct_ofsm_table M q1 q2`

Returns a minimal `n` such that states `q1`, `q2` reside in different classes in `(ofsm_table M f n)` where `f` is a function assigning to every state the set of states of `M`. If `M` is not observable or no such `n` exists, then the result is not specified.

`find_first_distinct_ofsm_table_gt M q1 q2 k`

Returns a minimal `n` such that `k` $\leq$ `n` and states `q1`, `q2` reside in different classes in `(ofsm_table M f n)` where `f` is a function assigning to every state the set of states of `M`. If `M` is not observable or no such `n` exists, then the result is not specified.

`get_distinguishing_sequence_from_ofsm_tables M q1 q2`

Returns an IO-trace `io` that distinguishes `q1`, `q2` in `M`. If `M` is not observable and minimal, or if no such `io` exists, then the result is not specified.

`select_diverging_ofsm_table_io M q1 q2 k`

Returns an IO-pair `(x,y)` that either immediately distinguishes states `q1`, `q2` in `M` or that, if applied to `q1`, `q2`, reaches states that reside in distinct classes in the `(k-1)`-th OFSM-Table. If `M` is not observable or no such `(x,y)` exists, then the result is not specified.

D.7 Convergence.thy

`converge M` α β

Checks whether α and β converge in `M`.

`preserves_converge M1 M2 TS`

Checks whether `TS` is `{M2}`-convergence-preserving w.r.t. `M1`.

`preserves_diverge M1 M2 TS`

Checks whether `TS` is `{M2}`-divergence-preserving w.r.t. `M1`.

`transition_cover M TS`

Checks whether the traces in `TS` constitute a state cover of OFSM `M`.

D.8 Convergence_Graph.thy

`convergence_graph_initial_invar M1 M2 lookup initial`

Checks whether the convergence graph obtained by applying to function `initial` an arbitrary set of traces (given as a finite `prefix_tree`), on which the languages of OFSMs `M1`, `M2` agree, satisfies the invariant on lookups performed by function `lookup`:

$$\forall\ \mathtt{T}.\ \mathcal{L}(\mathtt{M1}) \cap \mathtt{T} = \mathcal{L}(\mathtt{M2}) \cap \mathtt{T} \wedge \mathtt{finite\_tree\ T} \longrightarrow \mathtt{convergence\_graph\_lookup\_invar\ M1\ M2\ lookup\ (initial\ M1\ T)}$$

`convergence_graph_insert_invar M1 M2 lookup insert`

Checks whether the convergence graph obtained by inserting into an arbitrary convergence graph `G` a trace $\alpha \in \mathcal{L}(\mathtt{M1}) \cap \mathcal{L}(\mathtt{M2})$, for OFSMs `M1`, `M2`, via function `insert` satisfies the invariant on lookups performed by function `lookup`, assuming that `G` already satisfies this invariant:

$$\forall\ \mathtt{G}, \alpha.\ \alpha \in \mathcal{L}(\mathtt{M1}) \cap \mathcal{L}(\mathtt{M2}) \wedge \mathtt{convergence\_graph\_lookup\_invar\ M1\ M2\ lookup\ G} \longrightarrow \mathtt{convergence\_graph\_lookup\_invar\ M1\ M2\ lookup\ (insert\ G\ }\alpha\mathtt{)}$$

`convergence_graph_lookup_invar M1 M2 lookup G`

Checks whether the traces obtained by applying function `lookup` to convergence graph `G` and some trace $\alpha \in \mathcal{L}(\mathtt{M1}) \cap \mathcal{L}(\mathtt{M2})$ converge in both OFSMs `M1` and `M2`, while also returning α:

$$\forall \alpha \in \mathcal{L}(\mathtt{M1}) \cap \mathcal{L}(\mathtt{M2}).\ \alpha \in \mathtt{set\ (lookup\ G\ }\alpha\mathtt{)} \wedge \forall \beta \in \mathtt{set\ (lookup\ G\ }\alpha\mathtt{)}.\ \beta \in [\alpha]^{\mathtt{M1}} \cap [\alpha]^{\mathtt{M2}}$$

`convergence_graph_merge_invar M1 M2 lookup merge`

Checks whether the convergence graph obtained by merging in graph `G` via function `merge` some traces α, β that respectively converge in OFSMs `M1`

and M2 satisfies the invariant on lookups performed by function `lookup`, assuming that `G` already satisfies this invariant:

$$\forall\, \texttt{G}, \alpha, \beta.\, \beta \in [\alpha]^{\texttt{M1}} \wedge \beta \in [\alpha]^{\texttt{M2}} \\ \wedge\, \texttt{convergence\_graph\_lookup\_invar M1 M2 lookup G} \\ \longrightarrow\, \texttt{convergence\_graph\_lookup\_invar M1 M2 lookup} \\ \texttt{(merge G } \alpha\ \beta\texttt{)}$$

D.9 Empty_Convergence_Graph.thy

`empty_cg_empty`

Returns the empty list `[]`.

`empty_cg_initial M T`

Returns the empty list `[]`.

`empty_cg_insert G` α

Returns the empty list `[]`.

`empty_cg_lookup G` α

Returns `[`α`]`.

`empty_cg_merge G` α β

Returns the empty list `[]`.

D.10 Simple_Convergence_Graph.thy

In the following, I omit several functions that serve only as helper functions in implementing steps of `simple_cg_merge`.

`simple_cg_empty`

Returns the empty list `[]`.

`simple_cg_initial M T`

Returns the convergence graph obtained by iteratively inserting all traces in `T` that are in $\mathcal{L}$(`M`) into the empty convergence graph (that is, into `simple_cg_empty`).

`simple_cg_insert' G` α

Returns `G` if it already contains α. Otherwise adds the singleton set $\{\alpha\}$ to list `G`.

`simple_cg_insert G` α

Returns the convergence graph obtained by iteratively adding to `G` all prefixes of α via function `simple_cg_insert'`.

`simple_cg_lookup G` α

Returns a list containing α and all β such that $\{\alpha, \beta\}$ is a subset of some element in `G`.

`simple_cg_lookup_with_conv G` α

Returns a list containing α and all $\beta.\alpha''$ such that there exist some α' satisfying $\alpha = \alpha'.\alpha''$ and $\beta \in$ `List.set (simple_cg_lookup G` α'`)`.

`simple_cg_merge G` α β

Returns a convergence graph obtained by merging the sets containing α and β in `G`, followed by the closure operation described in [103]. See Subsection 9.4.3 for further details.

D.11 H_Framework.thy

`h_framework M f1 f2 f3 f4 f5 cgf1 cgf2 cgf3 cgf4 m`

Implementation of Algorithm 17 extended with functions `cg1` to `cgf4` for initialisation, insertion, lookup, and merge operations on convergence graphs. See Appendix E for the full definition.

`handles_io_pair f5 M I cgf2 cgf3`

Formalisation of condition ϕ_5^{H} of Lemma 6.3.1, extended with explicit handling of insertion and lookup operations on convergence graphs, as well as an additional requirement on the finiteness of the returned test suite.

`handles_transition f4 M I V TS cgf2 cgf3 cgf4`

Formalisation of condition ϕ_4^{H} of Lemma 6.3.1, extended with explicit handling of initialisation, insertion, and lookup operations on convergence graphs, as well as an additional requirement on the finiteness of the returned test suite.

`satisfies_abstract_h_condition M I V m`

Checks whether the abstract H-Condition (Lemma 4.1.7) holds for minimal reference model `M`, implementation I (both are assumed to be observable and to share the same alphabets), state cover assignment `V` of `M`, and upper bound `m` on the number of states in both `M` and `I`. See Subsection 10.1.1 for the full definition.

`separates_state_cover f3 M I cgf1 cgf2 cgf3`

Formalisation of condition ϕ_2^{H} of Lemma 6.3.1, extended with explicit handling of initialisation, insertion, and lookup operations on convergence graphs, as well as an additional requirement on the finiteness of the returned test suite.

D.12 SPY_Framework.thy

`spy_framework M f1 f2 f3 f4 f5 cgf1 cgf2 cgf3 cgf4 m`

Implementation of Algorithm 30 extended with functions `cg1` to `cgf4` for initialisation, insertion, lookup, and merge operations on convergence graphs.

`verifies_io_pair f5 M I cgf2 cgf3`

Formalisation of condition ϕ_5^{SPY} of Lemma 6.4.1, extended with explicit handling of insertion and lookup operations on convergence graphs, as well as an additional requirement on the finiteness of the returned test suite.

`verifies_transition f4 M I V TS cgf2 cgf3`

Formalisation of condition ϕ_4^{SPY} of Lemma 6.4.1, extended with explicit handling of insertion and lookup operations on convergence graphs, as well as a requirement on the finiteness of the returned test suite.

D.13 Pair_Framework.thy

`h_extensions M q k`

Computes set of traces X_q as used in Lemma 4.1.3 for arbitrary lengths `k` instead of the fixed $m - n$.

`language_up_to_length_with_extensions q hM iM ex k`

Computes the set of traces containing `ex` and all traces $\gamma.(x/y)$ such that $|\gamma| \leq$ `k` and $\gamma.(x/y)$ is in the language of `q` using transition relation `hM` and inputs `iM`.

`pair_framework M m f1 f2 f3`

Implementation of Algorithm 33.

`pairs_to_distinguish M V X Q`

Computes the sets A, B, C of Lemma 4.1.3 restricted to state set `Q` for FSM `M`, state cover assignment `V`, and function `X` such that `(X q)` returns set X_q of Lemma 4.1.3. To each trace its reached state in `M` is added.

`paths_up_to_length_with_targets q hM iM k`

Computes the set of all (p, q') such that p is a path from `q` of length at most `k` that reaches q', using only given transition relation `hM` and given inputs `iM`.

`satisfies_h_condition M V TS m`

Checks whether test suite `TS` contains sufficient traces to satisfy the weakened H-Condition (Lemma 4.1.3) for minimal OFSM `M`, state cover assignment `V`, and upper bound `m`.

D.14 Intermediate_Implementations.thy

In the following, parameter names `cgI`, `cgL` and `cgM` are employed to represent the insertion, lookup, and merge operations on convergence graphs. Furthermore, `compl` represents a flag whether certain input traces are to be completed, and `appH` represents the heuristic to employ in selecting convergent traces to append some trace to. Finally, `getDist` represents a procedural parameter for functions to compute distinguishing traces for given state pairs.

`add_distinguishing_sequence M ((io1,q1), (io2,q2)) t`

Implementation of GETDISTTRACE (Algorithm 37). For the supplied `(io,q)` pairs, state `q` is assumed to be reached by applying trace `io` to OFSM `M`.

`append_heuristic_input TS w (uBest,lBest) u'`

Performs the same comparisons as `append_heuristic_io`, but considers trace $\bar{x}/\bar{y}$ to be contained in `TS` as soon as `TS` contains some $\bar{x}/\bar{y}'$ with the same input portion.

`append_heuristic_io TS w (uBest,lBest) u'`

Heuristic to determine whether to append `w` in `TS` after `uBest` (assumed to require `lBest` new edges in the test suite tree) or `u'` by considering only the IO-traces currently in `TS`. Trace `u'` is preferred to `uBest` if `u'.w` $\in$ *pref*(`TS`), if appending `w` after `u'` would add fewer new branches to the test suite tree than adding after `u'`, or if both add the same number of new branches but adding after `u` requires fewer new edges.

`complete_inputs_to_tree M q ys xs`

Returns the set of all traces over the alphabets of `M` whose input portion is a prefix of `xs`, whose proper prefixes are in $\mathcal{L}_{\texttt{M}}(\texttt{q})$, and whose output portion contains only elements of `ys`.

`complete_inputs_to_tree_initial M xs`

Returns the set of all traces over the alphabets of `M` whose input portion is a prefix of `xs` and whose proper prefixes are in $\mathcal{L}$(`M`).

`contains_distinguishing_trace M TS q1 q2`

Checks whether *pref*(`TS`) contains a distinguishing trace for states `q1` and `q2`.

```
distinguish_from_set M V TS G cgL cgI getDist u v X d k
                     compl appH u_is_v
```

Implementation of DISTINGUISHFROMSET (Algorithm 12). Boolean flag `u_is_v` is used to indicate whether `u` and `v` are the same trace, in which case operations on `v` that have already performed on `u` are not repeated on `v`. Most procedural parameters are passed to `spyh_distinguish` and `distribute_extension`.

`distinguishing_set M`

Implementation of GETCHARSET (Algorithm 23).

`distinguishing_set_reduced M`

Implementation of GETCHARSET-REDUCED (Algorithm 41).

`distribute_extension M TS G cgL cgI u w compl appH`

Implementation of DISTRIBUTEEXTENSION (Algorithm 1). Selects some u' convergent with `u` using `appH` and appends `w` to it in the test suite `TS`. If `compl` is set, then the input portion of u'.`w` is completed in the test suite.

```
establish_convergence_dynamic compl useInputHeuristic
                              getDist M1 V TS G cgI cgL m t
```

Implementation of ESTABLISHCONV-DYNAMIC (Algorithm 32). If flag `useInputHeuristic` is set, then internal calls of `distribute_extension` use `append_heuristic_input` for procedural parameter `appH`. Otherwise, `append_heuristic_io` is employed.

`establish_convergence_static getDist M V TS G cgI cgL m t`

Implementation of ESTABLISHCONV-STATIC (Algorithm 31).

`estimate_growth M dist_fun q1 q2 x y errorValue`

Implementation of ESTIMATEGROWTH (Algorithm 16), using parameter `errorValue` instead of fixed return value $2|Q|$.

`get_HSI M q`

Implementation of getHSI (Algorithm 22).

`get_initial_test_suite_H V M m`

Implementation of getInitialTestSuite-H (Algorithm 34).

`get_initial_test_suite_H_2 c V M m`

Implementation of getInitialTestSuite-H (Algorithm 34) with an additional parameter `c`. Initially computes the same test suite as function `get_initial_test_suite`. If flag `c` is set, then this test suite is furthermore completed via `complete_inputs_to_tree_initial`.

`get_pairs_H V M m`

Implementation of getPairs-H (Algorithm 35).

`get_prefix_of_separating_sequence M TS G cgL getDist u v d`

Implementation of bestPrefixOfSepSeq (Algorithm 13). Performs at most `d` recursive steps to ensure termination for arbitrary convergence graphs.

```
handle_io_pair compl useInputHeuristic M V TS G
               cgI cgL q x y
```

Implementation of handleUndefinedIOPair (Algorithm 29) as well as handleIOPair (of the SPY-Framework, see Section 6.4). Parameter `useInputHeuristic` is employed in the same way as in function `establish_convergence_dynamic`.

```
handle_state_cover_dynamic compl useInputHeuristic getDist
                           M V cg_initial cgI cgL
```

Implementation of handleStateCover-Dynamic (Algorithm 19). Parameter `cg_initial` represents the initialisation operation on convergence graphs. Parameter `useInputHeuristic` is used in the same way as in `establish_convergence_dynamic`.

```
handle_state_cover_static get_dist_set M V cg_initial
                          cgI cgL
```

Implementation of handleStateCover-Static (Algorithm 20). Parameter `cg_initial` represents the initialisation operation on convergence graphs, while `dist_set` represents procedural parameter getDistinguishingSet of Algorithm 20.

`handleUT_dynamic compl useInputHeuristic getDist doEstablishConvergence M V TS G cgI cgL cgM m t X`

Implementation of HANDLEUT-DYNAMIC (Algorithm 27). Parameter `X` represents the set of remaining unverified transitions, which is used in the $S_{partial}$-Method to decide whether to establish convergence for transition `t`. This decision is performed by function `doEstablishConvergence`. Parameter `useInputHeuristic` is employed in the same way as in function `establish_convergence_dynamic`.

`handleUT_static dist_fun M V TS G cgI cgL cgM m t X`

Implementation of HANDLEUT-STATIC (Algorithm 25). Parameter `X` represents the set of remaining unverified transitions (unused by test strategies currently implemented using this function).

`has_extension TS G cgL io x y`

Implementation of HASEXTENSION (Algorithm 15).

`has_leaf TS G cgL io`

Implementation of HASLEAF (Algorithm 14).

`intersection_is_distinguishing M TS1 q1 TS2 q2`

Checks whether the intersection $pref(\texttt{TS1}) \cap pref(\texttt{TS2})$ contains a distinguishing trace for states `q1` and `q2`.

`shortest_list_in_tree_or_default xs TS x`

Returns the shortest list in $\{\texttt{x}\} \cup (\texttt{xs} \cap \texttt{TS})$.

`sort_unverified_transitions_by_state_cover_length M V ts`

Implementation of SPYH-SORTTRANSITIONS (Algorithm 24).

`spyh_distinguish M TS G cgL cgI getDist u X d compl appH`

Implements SPYH-DISTINGUISH (Algorithm 9). Establishes preservation of divergence over $\{\texttt{u}\} \cup \texttt{X}$, using `getDist` to select the required distinguishing traces. Most procedural parameters are passed on to functions `distribute_extension` and `get_prefix_of_separating_sequence`.

D.15 Intermediate_Frameworks.thy

In the following, `getDistSet` represents a function for computing harmonised state identifiers, while `getDist` represents a function returning a set containing at least one distinguishing trace for a given pair of states. Furthermore, `compl` represents a flag whether certain input traces are to be completed. Note that for each of the following functions, `Intermediate_Frameworks.thy` also contains a function of the same name extended with suffix `_lists`. These return test suites represented as lists of traces.

`h_framework_static_with_empty_graph M1 getDistSet m`

> Intermediate framework supporting implementation of the W, Wp, and HSI-Methods via the H-Framework. See Section 11.2.

`h_framework_static_with_simple_graph M1 getDistSet m`

> Intermediate framework supporting implementation of the SPY-Method via the H-Framework. See Section 11.2.

`h_framework_dynamic doEstablishConvergence M1 m compl useInputHeuristic`

> Intermediate framework supporting implementation of the H, SPYH, and $S_{partial}$-Methods via the H-Framework. See Section 11.2. Parameter `useInputHeuristic` is passed to function `handleUT_dynamic` (see Appendix D.14).

`pair_framework_h_components M m getDist`

> Intermediate framework supporting implementation of the W, HSI, and H-Methods via the Pair-Framework. In order to initialise the test suite, `get_initial_test_suite_H` (see Appendix D.14) is employed.

`pair_framework_h_components_2 M m getDist c`

> Intermediate framework supporting implementation of the W, HSI, and H-Methods via the Pair-Framework. In order to initialise the test suite, `get_initial_test_suite_H_2` (see Appendix D.14) is employed.

`spy_framework_static_with_empty_graph M1 getDistSet m`

> Intermediate framework supporting implementation of the W, Wp, and HSI-Methods via the SPY-Framework.

`spy_framework_static_with_simple_graph M1 getDistSet m`

> Intermediate framework supporting implementation of the SPY-Method via the SPY-Framework.

D.16 Wp_Method_Implementations.thy

`distinguishing_set_or_state_identifier l M k q`

Implementation of getCharSetOrHSI (Algorithm 26).

D.17 H_Method_Implementations.thy

In the following, `getDist` represents a procedural parameter for functions to compute distinguishing traces for given state pairs.

```
add_cheapest_distinguishing_trace getDist compl M
                                  ((io1,q1), (io2,q2)) TS
```

Similar to getDistTraceIfReq (Algorithm 38), but employs a heuristic function `find_cheapest_distinguishing_trace` to choose a distinguishing trace if `TS` does not already separate `io1` and `io2`, instead of simply using the result of getDistTrace (Algorithm 37). States `q1` and `q2` are assumed to be the states reached in OFSM `M` via `io1` and `io2`, respectively. If flag `compl` is set and a trace is to be added to test suite `TS`, then the completions of that trace derived via `complete_inputs_to_tree` (see Section D.14) for `q1` and `q2` are also inserted.

```
add_distinguishing_sequence_and_complete_if_required
    getDist compl M ((io1,q1), (io2,q2)) TS
```

Extends function `add_distinguishing_sequence_if_required` by completing inserted traces. That is, if $\bar{x}/\bar{y}$ is to be added to `TS`, then so too are all traces whose input portion is a prefix of $\bar{x}$ and that are in the language of `q1` or `q2`.

```
add_distinguishing_sequence_if_required
    getDist M ((io1,q1), (io2,q2)) TS
```

Implementation of getDistTraceIfReq (Algorithm 38). States `q1` and `q2` are assumed to be the states reached in OFSM `M` via `io1` and `io2`, respectively.

```
find_cheapest_distinguishing_trace
    M getDist ios TS1 q1 TS2 q2
```

Heuristic for selecting a distinguishing trace of `q1` and `q2` to be added. `TS1` and `TS2` represent the subtrees into both of which the trace is to be inserted, and `ios` represents the IO-pairs from which distinguishing traces are to be constructed. The selection is similar to that performed in function bestPrefixOfSepSeq (Algorithm 13), but does not employ convergence graphs, as it is to be employed in the Pair-Framework.

D.18 Partial_S_Method_Implementations.thy

`distance_at_most M q1 q2 k`

Checks whether there exists in OFSM `M` a path from state `q1` to state `q2` of length at most `k`.

`do_establish_convergence M V t X l`

Implementation of S-Heuristic (Algorithm 28). Parameter `X` represents the remaining unverified transitions.

Appendix E

H-Framework Implementation

The H-Framework (Algorithm 17, see Sections 6.3 and 10.1) is implemented in function `h_framework` of theory file `H_Framework.thy` as follows:

```
"h_framework M
             get_state_cover  handle_state_cover  sort_transitions
             handle_unverified_transition  handle_unverified_io_pair
             cg_initial  cg_insert  cg_lookup  cg_merge
             m
= (let
    rstates_set = reachable_states M;
    rstates     = reachable_states_as_list M;
    rstates_io  = product rstates (product (inputs_as_list M)
                                           (outputs_as_list M));
    undefined_io_pairs = filter (λ (q,(x,y)) . h_obs M q x y = None)
                                rstates_io;
    V   = get_state_cover M;
    TSG1 = handle_state_cover M V cg_initial cg_insert cg_lookup;
    sc_covered_transitions =
        (⋃ q ∈ rstates_set . covered_transitions M V (V q));
    unverified_transitions =
        sort_transitions M V (filter (λt . t_source t ∈ rstates_set
                                         ∧ t ∉ sc_covered_transitions)
                                     (transitions_as_list M));
    verify_transition =
        (λ (X,TS,G) t . handle_unverified_transition M V TS G
                            cg_insert cg_lookup cg_merge m t X);
    TSG2 = snd (foldl verify_transition (unverified_transitions, TSG1)
                                        unverified_transitions);
    verify_undefined_io_pair =
        (λ TS (q,(x,y)) . fst (handle_unverified_io_pair M V TS
                               (snd TSG2) cg_insert cg_lookup q x y))
  in
    foldl verify_undefined_io_pair (fst TSG2) undefined_io_pairs)"
```

This extends parameters of the pseudocode implementation given in Algorithm 17 by four functions to create, query and modify convergence graphs (see Chapter 9), namely `cg_initial`, `cg_insert`, `cg_lookup`, and `cg_merge`.

Following the parameters, the above implementation translates the pseudocode of Algorithm 17 into Isabelle/HOL, using a `let`-statement to introduce names for intermediate results, for example storing the state-input-output triples required in line 7 of Algorithm 17 as `undefined_io_pairs`. Thus, line 1 of Algorithm 17 is realised by the assignment to `V`, line 2 is realised by the creation of pair `TSG1`, and `unverified_transitions` corresponds to U after line 4 of Algorithm 17. Finally, the two loops in lines 5 and 7 are implemented by folding the application of `handle_unverified_transition` and `handle_unverified_io_pair`, respectively, over the corresponding lists `unverified_transitions` and `undefined_io_pairs`. This is performed via predefined function `foldl`, where `(foldl f y xs)` iteratively applies its procedural parameter `f` to the elements of a given list `xs` and the result of previous iterations, using `y` as initial value in to apply to the first element of `xs`. For example, `(foldl f y [x1,x2,x3])` evaluates to `(f (f (f y x1) x2) x3)`.

The Isabelle/HOL implementation of the H-Framework thus differs from Algorithm 17 predominantly in the explicit handling of operations on convergence graphs as procedural parameters. Furthermore, procedural parameter `handle_unverified_transition` uses an additional parameter in the list of transitions `X`. This parameter is employed in tracking the remaining unverified transitions required by the S-HEURISTIC of the $S_{partial}$-Method, which decides whether convergence is to be established for a transition (see Algorithm 28). This parameter has been omitted in Section 6.3 in order to simplify the presentation. It introduces no change in the conditions described in Lemma 6.3.1, as HANDLEUT-DYNAMIC satisfies ϕ_4^{H} for both possible outputs of the heuristic.