Action Codes

Frits Vaandrager
Radboud University, Nijmegen, The Netherlands

Thorsten Wißmann
Radboud University, Nijmegen, The Netherlands
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Abstract

We provide a new perspective on the problem how high-level state machine models with abstract actions can be related to low-level models in which these actions are refined by sequences of concrete actions. We describe the connection between high-level and low-level actions using *action codes*, a variation of the prefix codes known from coding theory. For each action code $\mathcal{R}$, we introduce a *contraction* operator $\alpha_{\mathcal{R}}$ that turns a low-level model $\mathcal{M}$ into a high-level model, and a *refinement* operator $\varrho_{\mathcal{R}}$ that transforms a high-level model $\mathcal{N}$ into a low-level model. We establish a Galois connection $\varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \Leftrightarrow \mathcal{N} \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{M})$, where $\sqsubseteq$ is the well-known simulation preorder. For conformance, we typically want to obtain an overapproximation of model $\mathcal{M}$. To this end, we also introduce a *concretization* operator $\gamma_{\mathcal{R}}$, which behaves like the refinement operator but adds arbitrary behavior at intermediate points, giving us a second Galois connection $\alpha_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{N} \Leftrightarrow \mathcal{M} \sqsubseteq \gamma_{\mathcal{R}}(\mathcal{N})$. Action codes may be used to construct adaptors that translate between concrete and abstract actions during learning and testing of Mealy machines. If Mealy machine $\mathcal{M}$ models a black-box system then $\alpha_{\mathcal{R}}(\mathcal{M})$ describes the behavior that can be observed by a learner/tester that interacts with this system via an adaptor derived from code $\mathcal{R}$. Whenever $\alpha_{\mathcal{R}}(\mathcal{M})$ implements (or conforms to) $\mathcal{N}$, we may conclude that $\mathcal{M}$ implements (or conforms to) $\gamma_{\mathcal{R}}(\mathcal{N})$.

Almost all results, examples, and counter-examples are formalized in Coq.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Automata, Models of Reactive Systems, LTS, Action Codes, Action Refinement, Action Contraction, Galois Connection, Model-Based Testing, Model Learning

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.137

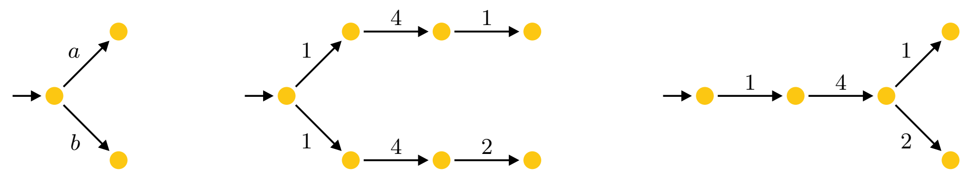
Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version With Appendix*: `https://arxiv.org/abs/2301.00199`

Supplementary Material *Software*: `https://gitlab.science.ru.nl/twissmann/action-codes-coq`, archived at `swh:1:dir:953b24c1e0771ce9ed7961f59f07294e0fd615d2`

Funding *Frits Vaandrager*: Supported by the NWO TOP project 612.001.852.
Thorsten Wißmann: Supported by the NWO TOP project 612.001.852 (until 2022) and the DFG project 419850228 (since 2023).

Acknowledgements As part of an MSc thesis project under supervision of the first author, Timo Maarse studied a different and more restricted type of action codes (called action refinements) [29]. It turned out, however, that for these action codes, the concretization operator is not monotone. The present paper arose from our efforts to fix this problem. We thank the anonymous reviewers for their suggestions, Paul Fiterău-Broştean for examples of the use of action codes in model learning, and Jules Jacobs for helpful discussions about Coq. The first author would like to thank Rocco De Nicola for his hospitality at IMT Lucca during the work on this paper.



(a) Original System. **(b)** Existing action refinements. **(c)** Desired action refinement behavior.

Figure 1 Example for the (lack of) preservation of determinism in action refinement.

1 Introduction

Labeled transition systems (LTSs) constitute one of the most fundamental modeling mechanisms in Computer Science. An LTS is a rooted, directed graph whose nodes represent *states* and whose edges are labeled with *actions* and represent *state transitions*. LTS-based formalisms such as Finite Automata [21], Finite State Machines [25], I/O automata [26], IOTSs [35], and process algebras [4] have been widely used to model and analyze a broad variety of reactive systems, and a rich body of theory has been developed for them.

In order to manage the complexity of computer-based systems, designers structure such systems into hierarchical layers. This allows them to describe and analyze systems at different levels of abstraction. Many LTS-based frameworks have been proposed to formally relate models at different hierarchical levels, e.g. [4, 14, 27, 40]. In most of these frameworks, the states of a high-level LTS correspond to sets of states of a low-level LTS via simulation or bisimulation-like relations. However, the actions are fixed and considered to be atomic. Actions used at a lower level of abstraction can be hidden at a higher level, but higher-level actions will always be available at the lower level. For this reason, Rensink & Gorrieri [18, 31] argue that these (bi)simulations relate systems at the same conceptual level of abstraction, and therefore they call them *horizontal* implementation relations. They contrast them with *vertical* implementation relations that compare systems that belong to conceptually different abstraction levels, and have different alphabets of actions.

A prototypical example of a hierarchical design is a computer network. To reduce design complexity, such a network is organized as a stack of layers or levels, each one built upon the one below it [34]. Examples are the transport layer, with protocols such as TCP and UDP, and the physical layer, concerned with transmitting raw bits over a communication channel. Now consider a host that receives a TCP packet in some state s. If P is the set of possible packets then, in an LTS model of the transport layer, state s will contain outgoing transitions labeled with action $receive(p)$, for each $p \in P$. At the physical layer, however, receipt of a packet corresponds to a sequence of $receive(b)$ actions, with b a bit in $\{0, 1\}$. Only after the final bits have arrived, the host knows which packet was actually received. Mechanisms for transforming high-level actions into sequences (or processes) of low-level actions have been addressed extensively in work on action refinements [18]. These approaches, however, are unable to describe the above scenario in a satisfactory manner and somehow assume that a host upfront correctly guesses the packet that it will receive, even before the first bit has arrived. In order to illustrate this problem, we consider the simplified example of an LTS with a distinguished initial state, displayed in Figure 1a, which accepts either input a or input b. At a lower level of abstraction, input a is implemented by three consecutive inputs 1 4 1, whereas input b is implemented by action sequence 1 4 2 (the ASCII encodings of a and b in octal format). An action refinement operator will replace the a-transition in Figure 1a by a sequence of three consecutive transitions with labels 1, 4 and 1, respectively, and will handle the b-transition in an analogous manner. Thus, action refinement introduces a nondeterministic choice (Figure 1b), rather than the deterministic behavior that one would like to see (Figure 1c). As a consequence of this and other limitations, refinement operators have not found much practical use [18].

Based on the observation that any action can be modeled as a state change, some authors (e.g. [2, 10, 24]) prefer modeling formalisms in which the term "action" is only used informally, and Kripke structures rather than LTSs are used to model systems. These state-based approaches have the advantage that a distinction between horizontal and vertical implementation relations is no longer needed, and a single implementation relation suffices. Purely state-based approaches, however, are problematic in cases where we need to interact with a black-box system and (by definition) we have no clue about the state of this system. Black-box systems prominently occur in the areas of model-based testing [36] and model learning [37]. In these application areas, use of LTSs makes sense and there is a clear practical need for formalisms that allow engineers to relate actions at different levels of abstraction.

Van der Bijl et al. [7], for instance, observe that in model-based testing specifications are usually more abstract than the System Under Test (SUT). This means that generated test cases may not have the required level of detail, and often a single abstract action has to be translated (either manually or by an adaptor) to a sequence of concrete actions that are applied to the SUT. Van der Bijl et al. [7] study a restricted type of action refinement in which a single input is refined into a sequence of inputs, and implement this in a testing tool.

Also in model learning, typically an adaptor is placed in between the SUT and the learner, to take care of the translation between abstract and concrete actions. For example, in a case study on hand-held smartcard readers for Internet banking, Chalupar et al. [9] used abstract inputs that combine several concrete inputs in order to accelerate the learning process and reduce the size of the learned model. In particular, they introduced a single abstract input COMBINED_PIN corresponding to a USB command, followed by a 4-digit PIN code, followed by an OK command. Fiterău-Broştean et al. [12] used model learning for a comprehensive analysis of DTLS implementations, and found four serious security vulnerabilities, as well as several functional bugs and non-conformance issues. Handshakes in (D)TLS are defined over flights of messages. Hence, (D)TLS entities are often expected to produce multiple messages before expecting a response. During learning, Fiterău-Broştean et al. [12] used an adaptor that contracted multiple messages from the SUT into a single abstract output. Also in other case studies on TLS [32], Wi-Fi [33] and SSH [39, 13], multiple outputs from the SUT were contracted into a single abstract output. Verleg [39] used a single abstract input to execute the entire key re-exchange during learning higher layers of SSH.

In this article, we provide answers to two fundamental questions: (1) How can we formalize the concept of an *adaptor* that translates between abstract and concrete actions?, and (2) Suppose the behavior of an SUT is described by an unknown, concrete model $\mathcal{M}$, and suppose a learner interacts with this SUT through an adaptor and learns an abstract model $\mathcal{N}$. What can we say about the relation between $\mathcal{M}$ and $\mathcal{N}$?

We answer the first question by introducing *action codes*, a variation of the prefix codes known from coding theory [5]. Action codes describe how high-level actions are converted into sequences of low-level actions, and vice versa. This makes them different from action refinements, which specify how high-level actions can be translated into low-level processes, but do not address the reverse translation. Our notion of an action code captures adaptors that are used in practice, and in particular those described in the case studies listed above.

In order to answer the second question we introduce, for each action code $\mathcal{R}$, a *contraction* operator $\alpha_{\mathcal{R}}$ that turns a low-level model $\mathcal{M}$ into a high-level model by contracting concrete action sequences of $\mathcal{M}$ according to $\mathcal{R}$. We also introduce the left adjoint of $\alpha_{\mathcal{R}}$, the *refinement* operator $\varrho_{\mathcal{R}}$ that turns a high-level model $\mathcal{M}$ into a low-level model by refining abstract actions of $\mathcal{N}$ according to $\mathcal{R}$. This refinement operator, for instance, maps the LTS of Figure 1a to the LTS of Figure 1c. We establish a Galois connection $\varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \Leftrightarrow \mathcal{N} \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{M})$, where $\sqsubseteq$ denotes the simulation preorder. So if an abstract model $\mathcal{N}$ implements contraction $\alpha_{\mathcal{R}}(\mathcal{M})$, then the refinement $\varrho_{\mathcal{R}}(\mathcal{N})$ implements concrete model $\mathcal{M}$, and vice versa.

In practice, we typically want to obtain an overapproximation of concrete model $\mathcal{M}$. To this end, we introduce the right adjoint of $\alpha_{\mathcal{R}}$, the *concretization* operator $\gamma_{\mathcal{R}}$. This operator behaves like the refinement operator, but adds arbitrary behavior at intermediate points (cf. the demonic completion of [6]). We establish another Galois connection: $\alpha_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{N} \Leftrightarrow \mathcal{M} \sqsubseteq \gamma_{\mathcal{R}}(\mathcal{N})$. This connection is useful, because whenever we have established that $\alpha_{\mathcal{R}}(\mathcal{M})$ implements (or conforms to) $\mathcal{N}$, it allows us to conclude that $\mathcal{M}$ implements (or conforms to) $\gamma_{\mathcal{R}}(\mathcal{N})$.

We show that, in a setting of Mealy machines (subsuming Finite State Machines), an *adaptor* can be constructed for any action code for which a winning strategy exists in a certain 2-player game. If a learner/tester interacts with an SUT via an adaptor generated from such an action code $\mathcal{R}$, and the SUT is modeled by Mealy machine $\mathcal{M}$, then from the learner/tester perspective, the composition of adaptor and SUT will behave like $\alpha_{\mathcal{R}}(\mathcal{M})$. Thus, if a learner succeeds to learn an abstract model $\mathcal{N}$ such that $\mathcal{N} \approx \alpha_{\mathcal{R}}(\mathcal{M})$ then, using the Galois connections, the learner may conclude that $\varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \sqsubseteq \gamma_{\mathcal{R}}(\mathcal{N})$.

The remainder of this article is structured as follows. We start with a preliminary Section 2 that introduces basic notations and results for LTSs. Next, action codes and the contraction operator are introduced in Section 3. After describing the refinement operator, we establish our first Galois connection in Section 4. Next we define concretization and establish our second Galois connection in Sections 5. Section 6 explains how action codes can be composed, and shows that contraction and refinement commute with action code composition. Section 7 describes how adaptors can be constructed from action codes. Finally, Section 8 contains a discussion of our results and identifies directions for future research.

Almost all proofs are formalized in Coq (about 6000 lines of code) and can be accessed via `https://gitlab.science.ru.nl/twissmann/action-codes-coq` and via the ancillary files of the full version on arxiv. We mark formalized results with a clickable Coq icon pointing to the respective location in the HTML documentation. Appendix A (in the full version) contains comments on the Coq formalization and Appendix B contains full proofs (in natural language) and additional remarks.

2 Preliminaries

If Σ is a set of symbols then Σ^* denotes the set of all finite words over Σ, and Σ^+ the set of all non-empty words. We use ε to denote the empty word, so e.g. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Concatenation of words $u, w \in \Sigma^*$ is notated $u \cdot w$ (or simply $u\,w$). We write $u \leq w$ if u is a prefix of w, i.e. if there is $v \in \Sigma^*$ with $u\,v = w$. We write $|w|$ to denote the length of word w.

We use $f\colon X \rightharpoonup Y$ to denote a partial map f from X to Y and write $\mathsf{dom}(f) \subseteq X$ for its domain, i.e. set of $x \in X$ on which f is defined. The *image* $\mathsf{im}(f)$ of a partial map $f\colon X \rightharpoonup Y$ is the set of elements of Y it can reach: $\mathsf{im}(f) := \{f(x) \mid x \in \mathsf{dom}(f)\} \subseteq Y$.

▶ **Definition 2.1** (). *For a set A of action labels, a* labeled transition system (LTS) *is a tuple $\mathcal{M} = \langle Q, q_0, \longrightarrow \rangle$ where Q is a set of states, $q_0 \in Q$ is a starting state, and $\longrightarrow \subseteq Q \times A \times Q$ is a transition relation. We write $\mathsf{LTS}(A)$ for the class of all LTSs with labels from A. We refer to the three components of an LTS $\mathcal{M}$ as $Q^{\mathcal{M}}$, $q_0^{\mathcal{M}}$ and $\longrightarrow_{\mathcal{M}}$, respectively, and introduce the following notation:*

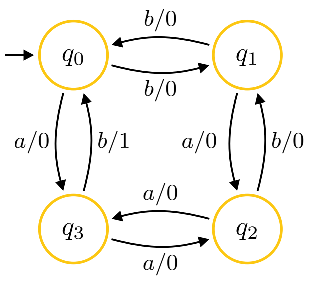


Figure 2 A Mealy machine.

- $q \xrightarrow{a} q'$ *denotes* $(q, a, q') \in \longrightarrow$; $q \xrightarrow{a}$ *denotes that there is some* q' *with* $q \xrightarrow{a} q'$;
- $q \xRightarrow{w} q'$ *for* $w \in A^*$ *denotes that there are finite sequences* $a_1, \ldots, a_n \in A$, $r_0, \ldots, r_n \in Q$ *such that* $w = a_1 \cdots a_n$, *and* $r_0 = q$, $r_n = q'$ *and* $r_{i-1} \xrightarrow{a_i} r_i$ *for all* $1 \le i \le n$;
- $q \xRightarrow{w}$ *denotes that there is* q' *such that* $q \xRightarrow{w} q'$;
- $q \in Q$ *is* reachable *if there is* $w \in A^*$ *such that* $q_0 \xRightarrow{w} q$.

A special class of LTSs that is frequently used in conformance testing and model learning are *Mealy machines*. Mealy machines with a finite number of states are commonly referred to as *Finite State Machines*.

▶ **Definition 2.2.** *For non-empty sets of inputs* I *and outputs* O, *a (non-deterministic)* Mealy machine $\mathcal{M} \in \mathsf{LTS}(I \times O)$ *is an LTS where the labels are pairs of an input and an output. We write* $q \xrightarrow{i/o} q'$ *to denote that* $(q, (i, o), q') \in \longrightarrow$. *Whenever we omit a symbol in predicate* $q \xrightarrow{i/o} q'$ *this is quantified existentially. Thus,* $\xrightarrow{i/o}$ *if there are* q *and* q' *s.t.* $q \xrightarrow{i/o} q'$, $q \xrightarrow{i/} q'$ *if there is an* o *s.t.* $q \xrightarrow{i/o} q'$, *and* $q \xrightarrow{i/}$ *if there is a* q' *s.t.* $q \xrightarrow{i/} q'$.

▶ **Example 2.3** (). Figure 2 visualizes a simple Mealy machine with inputs $\{a, b\}$ and outputs $\{0, 1\}$. The machine always outputs 0 in response to an input, except in one specific situation. Output 1 is produced in response to input b if the previous input was a and the number of preceding inputs is odd. The machine has four states q_0, q_1, q_2 and q_3, with starting state q_0 marked by an incoming arrow. In states q_0 and q_2 the number of preceding inputs is always even, whereas in states q_1 and q_3 it is always odd. In states q_2 and q_3 the previous input is always a, whereas in states q_0 and q_1 either the previous input is b, or no input has occurred yet. Thus, only in state q_3 input b triggers output 1.

We introduce some notation and terminology for LTSs.

▶ **Definition 2.4** (). *Let* $\mathcal{M} = \langle Q, q_0, \longrightarrow \rangle \in \mathsf{LTS}(A)$ *be an LTS. We say that*

- $\mathcal{M}$ *is* deterministic *if, whenever* $q \xrightarrow{a}$ *for some* q *and* a, *there is a unique* q' *with* $q \xrightarrow{a} q'$.
- $\mathcal{M}$ *is a* tree-shaped *if each state* $q \in Q$ *can be reached via a unique sequence of transitions from state* q_0.
- $q \in Q$ *is a* leaf, *notated* $q \not\longrightarrow$, *if there is no* $a \in A$ *with* $q \xrightarrow{a}$.
- $\mathcal{M}$ *is* grounded *if every state* $q \in Q$ *has a path to a leaf.*

We can now define the set of traces of an LTS:

▶ **Definition 2.5** (). *Let* $\mathcal{M} = \langle Q, q_0, \longrightarrow \rangle \in \mathsf{LTS}(A)$. *A word* $w \in A^*$ *is a* trace *of state* $q \in Q$ *if* $q \xRightarrow{w}$, *and a* trace *of* $\mathcal{M}$ *if it is a trace of* q_0. *We write* $trace(\mathcal{M})$ *for the set* $\{w \in A^* \mid q_0 \xRightarrow{w}\}$ *of all traces of* $\mathcal{M}$.

▶ **Definition 2.6** (Simulation,). *For $\mathcal{M}, \mathcal{N} \in \mathsf{LTS}(A)$, a* simulation *from $\mathcal{M}$ to $\mathcal{N}$ is a relation $S \subseteq Q^{\mathcal{M}} \times Q^{\mathcal{N}}$ such that*

1. *$q_0^{\mathcal{M}} \; S \; q_0^{\mathcal{N}}$ and*
2. *if $q_1 \; S \; q_2$ and $q_1 \xrightarrow{a}_{\mathcal{M}} q_1'$ then there exists a state q_2' such that $q_2 \xrightarrow{a}_{\mathcal{N}} q_2'$ and $q_1' \; S \; q_2'$.*

We write $\mathcal{M} \sqsubseteq \mathcal{N}$ if there exists a simulation from $\mathcal{M}$ to $\mathcal{N}$.

It is a classical result that trace inclusion coincides with the simulation preorder for deterministic labeled transition systems (see e.g. [28]):

▶ **Lemma 2.7** (). *For all $\mathcal{M}, \mathcal{N} \in \mathsf{LTS}(A)$ where $\mathcal{N}$ is deterministic: $trace(\mathcal{M}) \subseteq trace(\mathcal{N})$ iff $\mathcal{M} \sqsubseteq \mathcal{N}$.*

We will often consider LTSs up to isomorphism of their reachable parts:

▶ **Definition 2.8** (Isomorphism,). *For $\mathcal{M}, \mathcal{N} \in \mathsf{LTS}(A)$, an* isomorphism *from $\mathcal{M}$ to $\mathcal{N}$ is a bijection $f \colon Q_R^{\mathcal{M}} \to Q_R^{\mathcal{N}}$, where:*

1. *$Q_R^{\mathcal{M}} \subseteq Q^{\mathcal{M}}$ and $Q_R^{\mathcal{N}} \subseteq Q^{\mathcal{N}}$ are the subsets of reachable states in $\mathcal{M}$ and $\mathcal{N}$, respectively;*
2. *$f(q_0^{\mathcal{M}}) = q_0^{\mathcal{N}}$, and*
3. *$q \xrightarrow{a}_{\mathcal{M}} q'$ iff $f(q) \xrightarrow{a}_{\mathcal{N}} f(q')$, for all $q, q' \in Q_R^{\mathcal{M}}$, $a \in A$.*

We write $\mathcal{M} \cong \mathcal{N}$ if there exists an isomorphism from $\mathcal{M}$ to $\mathcal{N}$.

Note that $\cong$ is an equivalence relation on $\mathsf{LTS}(A)$, and that $\mathcal{M} \cong \mathcal{N}$ implies $\mathcal{M} \sqsubseteq \mathcal{N}$, since each isomorphism (when viewed as a relation) is trivially a simulation.

3 Action Codes

Adaptors that are used for learning and testing translate sequences of abstract actions into sequences of concrete actions, and vice versa. Action codes describe *how* an adaptor may translate between two action label alphabets, for example from A to B. Intuitively, we understand the first alphabet A as the actions at the lower, concrete level, and the second alphabet B as the actions at the higher, more abstract level. In an action code, a single abstract action $b \in B$ corresponds to a finite, non-empty sequence of concrete actions $a_1 \cdots a_n$ in A. Essentially, action codes are just a special type of *prefix codes* [5], as known from coding theory. Prefix codes have the desirable property that they are *uniquely decodable*: given a sequence of concrete actions, there is at most one corresponding sequence of abstract actions. We provide two equivalent definitions of action codes: one via tree-shaped LTSs and one via partial maps.

▶ **Definition 3.1** (Action code,). *For sets of action labels A and B, a* (tree-shaped) action code *$\mathcal{R}$ from A to B is a structure $\mathcal{R} = \langle \mathcal{M}, l \rangle$, with $\mathcal{M} = \langle R, r_0, \longrightarrow \rangle \in \mathsf{LTS}(A)$ a deterministic, tree-shaped LTS with L being the set of non-root leaves $L \subseteq R \setminus \{r_0\}$ and an injective function $l \colon L \to B$. We write $\mathsf{Code}(A, B)$ for all action codes from A to B.*

The injectivity of l and the tree-shape ensure that every abstract $b \in B$ is represented by at most one $w \in A^+$.

▶ **Example 3.2.** Figure 3 shows an action code for a fragment of the ASCII encoding in octal format, e.g., 1 1 5 encodes the letter `M`, 1 4 5 encodes the letter `e`, etc.

▶ **Example 3.3.** Figure 4 shows an action code for the activity of getting a cup of coffee or espresso, in the special case of Mealy machines, i.e. where $A = I \times O$ and $B = I' \times O'$ are sets of input/output-pairs. Rather than the full sequence of interventions that is required in order to get a drink, the abstract input/output pair only reports on the type of drink that was ordered and the number of interventions that occurred.

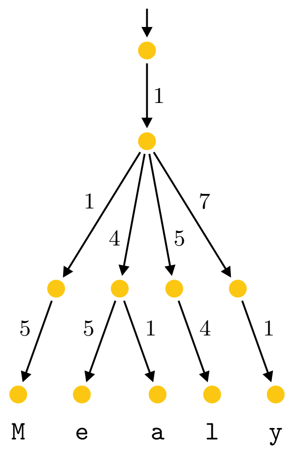


Figure 3 Action code for a fragment of ASCII.

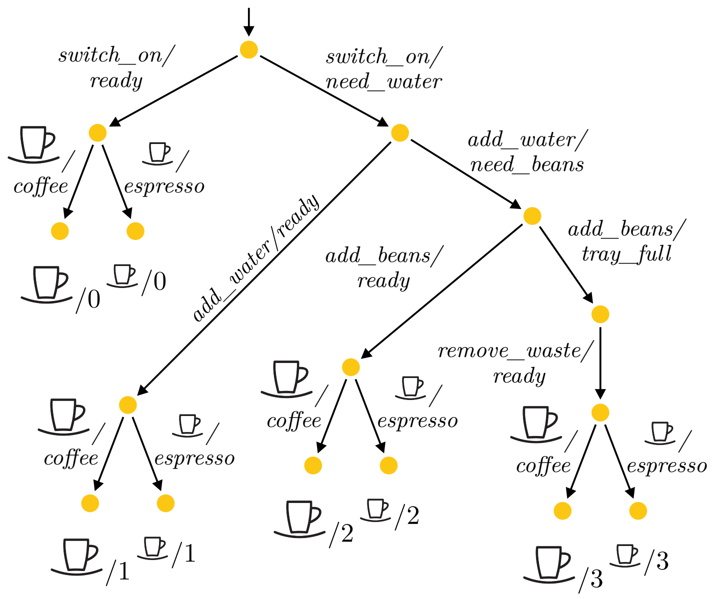


Figure 4 Action code for a coffee machine.

The definition of action codes as LTSs allows an intuitive visualization. For easier mathematical reasoning, we characterize action codes also in terms of maps:

▶ **Definition 3.4** (). *A* (map-based) *action code from A to B is a partial map $f\colon B \rightharpoonup A^+$ which is* prefix-free*, by which we mean that for all $b, b' \in \mathsf{dom}(f)$,*

$$f(b) \leq f(b') \qquad \textit{implies} \qquad b = b'. \tag{1}$$

In the following, we show that these prefix-free partial maps bijectively correspond to the tree-shaped LTSs:

▶ **Lemma 3.5** (). *Every tree-shaped action code $\mathcal{R} \in \mathsf{Code}(A, B)$ induces a unique map-based action code $f\colon B \rightharpoonup A^+$ with the property that for all $b \in B, w \in A^+$:*

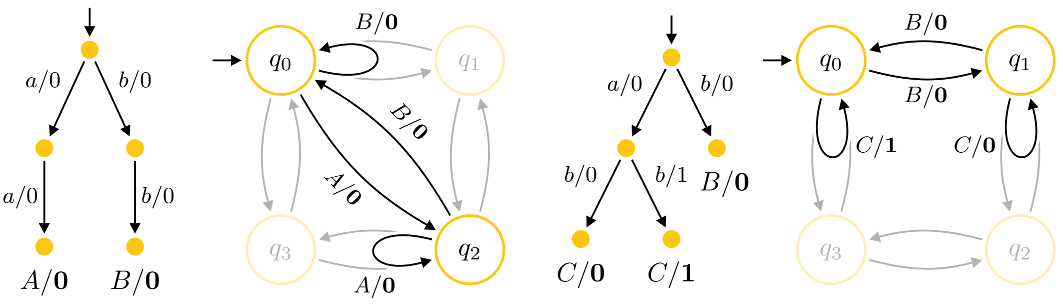
$$f(b) = w \qquad \textit{iff} \qquad \exists r \in L\colon r_0 \xRightarrow{w}_{\mathcal{R}} r, \quad l(r) = b \tag{2}$$

▶ **Lemma 3.6** (). *For each map-based action code $f\colon B \rightharpoonup A^+$, there is (up to isomorphism) a unique tree-shaped action code $\mathcal{R} \in \mathsf{Code}(A, B)$ which is grounded and satisfies* (2)*.*

▶ **Example 3.7** (). For the uniqueness in Lemma 3.6, we use groundedness, because for $A = \{a\}$ and any B, the action codes

$$\mathcal{R} := (\rightarrow\bullet \xrightarrow{a} \bullet \xrightarrow{a} \bullet \xrightarrow{a} \cdots) \quad \text{and} \quad \mathcal{S} := (\rightarrow\bullet).$$

both have no non-root leaves, and so they both induce the empty partial map $f\colon B \rightharpoonup A^+$ via Lemma 3.5. This f is undefined for all $b \in B$. And indeed, $\mathcal{R}$ and $\mathcal{S}$ are not isomorphic. The issue is that while the finite $\mathcal{S}$ is grounded, the infinite $\mathcal{R}$ is not grounded. So $\mathcal{R}$ contains subtrees which do not contribute anything to the partial map f but which hinder the existence of an isomorphism.



(a) An action code $\mathcal{R}$ together with $\alpha_{\mathcal{R}}(\mathcal{M})$. **(b)** Another code $\mathcal{S}$ together with $\alpha_{\mathcal{S}}(\mathcal{M})$.

Figure 5 The resulting contraction of the LTS $\mathcal{M}$ from Figure 2 for different action codes.

Having shown the correspondence between tree-shaped and map-based action codes $\mathsf{Code}(A, B)$, we can switch between the two views in proofs. Mostly, we use the tree-shaped version for visualization and the map-based version for mathematical reasoning.

Consider a concrete $\mathcal{M} \in \mathsf{LTS}(A)$, together with an action code $\mathcal{R}$ from A to B. We can construct an abstract LTS for the action labels B by walking through $\mathcal{M}$ with seven-league boots, repeatedly choosing input sequences that correspond to runs to some leaf of $\mathcal{R}$, and then contracting this sequence to a single abstract transition.

▶ **Notation 3.8.** In the rest of the paper, we introduce operators $\alpha_{\mathcal{R}}$, $\varrho_{\mathcal{R}}$, $\gamma_{\mathcal{R}}$ on LTSs, involving an action code $\mathcal{R}$. Whenever the action code $\mathcal{R}$ is clear from the context, we omit the index and simply speak of operators α, ϱ, γ for the sake of brevity.

▶ **Definition 3.9** (Contraction,). *For each action code $\mathcal{R} \in \mathsf{Code}(A, B)$, the* contraction *operator $\alpha_{\mathcal{R}} \colon \mathsf{LTS}(A) \to \mathsf{LTS}(B)$ is defined as follows. For $\mathcal{M} \in \mathsf{LTS}(A)$, the LTS $\alpha_{\mathcal{R}}(\mathcal{M})$ has states $Q^{\alpha(\mathcal{M})} \subseteq Q^{\mathcal{M}}$ and transitions $\xrightarrow{}_{\alpha(\mathcal{M})}$ defined inductively by the rules (1_α) and (2_α), for all $q, q' \in Q^{\mathcal{M}}$, $b \in B$.*

$$\frac{}{q_0^{\mathcal{M}} \in Q^{\alpha(\mathcal{M})}} \quad (1_\alpha) \qquad\qquad \frac{q \in Q^{\alpha(\mathcal{M})}, \quad b \in \mathsf{dom}(\mathcal{R}), \quad q \xRightarrow{\mathcal{R}(b)}_{\mathcal{M}} q'}{q \xrightarrow{b}_{\alpha(\mathcal{M})} q', \qquad q' \in Q^{\alpha(\mathcal{M})}} \quad (2_\alpha)$$

The initial state $q_0^{\alpha(\mathcal{M})} := q_0^{\mathcal{M}}$ is the same as in $\mathcal{M}$.

▶ **Example 3.10.** Figures 5 shows two examples of action codes and the contractions obtained when we apply them to the Mealy machine of Figure 2 (with the original machine shaded in the background). The examples illustrate that by choosing different codes we may obtain completely different abstractions of the same LTS.

The next proposition asserts that we can view $\alpha_{\mathcal{R}}$ as a monotone function $\alpha_{\mathcal{R}} \colon \mathsf{LTS}(A) \to \mathsf{LTS}(B)$ between preordered classes.

▶ **Proposition 3.11** (Monotonicity,). *For every action code $\mathcal{R} \in \mathsf{Code}(A, B)$, whenever $\mathcal{M} \sqsubseteq \mathcal{N}$ for $\mathcal{M}, \mathcal{N} \in \mathsf{LTS}(A)$, then $\alpha_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{N})$ in $\mathsf{LTS}(B)$.*

4 Refinements

Now that we have introduced the contraction $\alpha_{\mathcal{R}}$ of an LTS for a code $\mathcal{R}$, it is natural to consider an operation in the other direction, which we call the *refinement* $\varrho_{\mathcal{R}}$. Intuitively, refinement replaces each abstract transition $q \xrightarrow{b} q'$ by a sequence of concrete transitions, as prescribed by $\mathcal{R}$.

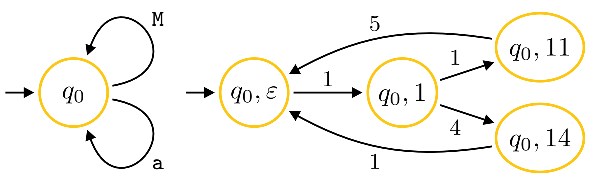


Figure 6 LTS and its refinement w.r.t $\mathcal{R}$ of Figure 3.

$$\begin{tikzcd} \varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \arrow[r, Rightarrow, shift left, "\text{If } \mathcal{N} \in \mathsf{LTS}(\mathsf{dom}(\mathcal{R}))"] & \mathcal{N} \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{M}) \arrow[l, Rightarrow, shift left, "\text{If } \mathcal{M} \text{ is deterministic}"] \end{tikzcd}$$

Figure 7 Theorem 4.5.

▶ **Definition 4.1** (Refinement,). *For each action code $\mathcal{R} \in \mathsf{Code}(A, B)$, we define the* refinement *operator $\varrho_{\mathcal{R}}\colon \mathsf{LTS}(B) \to \mathsf{LTS}(A)$ as follows. For $\mathcal{M} \in \mathsf{LTS}(B)$, the LTS $\varrho_{\mathcal{R}}(\mathcal{M}) \in \mathsf{LTS}(A)$ has a set of states*

$$Q^{\varrho(\mathcal{M})} := \{(q, w) \in Q^{\mathcal{M}} \times A^* \mid w = \varepsilon \text{ or (there is } b \text{ with } q \xrightarrow{b}_{\mathcal{M}} \text{ and } w \lneq \mathcal{R}(b))\}$$

and the initial state $(q_0^{\mathcal{M}}, \varepsilon)$. The transition relation $\longrightarrow_{\varrho(\mathcal{M})}$ is defined by the following rules:

$$\frac{(q, wa) \in Q^{\varrho(\mathcal{M})}}{(q, w) \xrightarrow{a}_{\varrho(\mathcal{M})} (q, wa)} \ (1_\varrho) \qquad\qquad \frac{q \xrightarrow{b}_{\mathcal{M}} q' \quad wa = \mathcal{R}(b)}{(q, w) \xrightarrow{a}_{\varrho(\mathcal{M})} (q', \varepsilon)} \ (2_\varrho)$$

Intuitively, whenever $\varrho(\mathcal{M})$ is in state (q, w), then this corresponds to being in state q in the abstract automaton $\mathcal{M} \in \mathsf{LTS}(B)$ and having observed the actions $w \in A^*$ so far. However, we have insufficiently many actions for finding an abstract transition $q \xrightarrow{b}_{\mathcal{M}} q'$ with $w = \mathcal{R}(b)$ because w is still to short. Nevertheless, whenever $\varrho(\mathcal{M})$ admits a transition to a state (q, w) with $w \neq \varepsilon$, then we know that we can eventually complete w to a sequence corresponding to an abstract transition: there exist at least one $q \xrightarrow{b}_{\mathcal{M}} q'$ for some $b \in \mathsf{dom}(\mathcal{R})$ with $w \leq \mathcal{R}(b)$. If the abstract system $\mathcal{M}$ is non-deterministic, then there may be multiple abstract transitions that match in the final rule (2_ϱ), but the transitions produced by rule (1_ϱ) are deterministic.

▶ **Example 4.2.** Figure 6 shows an example application of a refinement operator that replaces the actions of the LTS $\mathcal{M}$ on the left by their ASCII encoding in octal format, as prescribed by the action code from Figure 3. The initial state is (q_0, ε), corresponding to q_0 in $\mathcal{M}$. Since $\mathcal{M}$ contains abstract labels `M` and `a`, with $\mathcal{R}(\mathtt{M}) = 1\,1\,5$ and $\mathcal{R}(\mathtt{a}) = 1\,4\,1$, we need to introduce additional states for having read 1, $1\,1$, and $1\,4$, because those are the sequences of A-actions before we have observed a sequence $\mathcal{R}(b) \in A^+$ for some $b \in B$.

A more visual explanation of $\varrho_{\mathcal{R}}(\mathcal{M})$ is the following: for every state $q \in Q^{\mathcal{M}}$, we consider the outgoing transitions $\{q \xrightarrow{b}_{\mathcal{M}} q' \mid b \in B, q' \in Q^{\mathcal{M}}\}$ and labels $B' \subseteq B$ that appear in it. Then, this outgoing-transition structure is replaced with (a copy of) the minimal subgraph of the tree $\mathcal{R}$ containing all leaves with labels in B'.

Like contraction, the refinement operation also preserves the simulation preorder.

▶ **Proposition 4.3** (Monotonicity,). *For all action codes $\mathcal{R} \in \mathsf{Code}(A, B)$, if $\mathcal{M} \sqsubseteq \mathcal{N}$ in $\mathsf{LTS}(B)$, then $\varrho_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \varrho_{\mathcal{R}}(\mathcal{N})$ in $\mathsf{LTS}(A)$.*

As $\mathcal{R}$ is deterministic, applying $\varrho_{\mathcal{R}}$ on a deterministic LTS results in a deterministic LTS:

▶ **Proposition 4.4** (Refinement preserves determinism,). *For every action code $\mathcal{R} \in \mathsf{Code}(A, B)$, if $\mathcal{M} \in \mathsf{LTS}(B)$ is deterministic, then $\varrho_{\mathcal{R}}(\mathcal{M}) \in \mathsf{LTS}(A)$ is deterministic, too.*

▶ **Theorem 4.5** (Galois connection,). *For $\mathcal{R} \in \mathsf{Code}(A, B)$, $\mathcal{N} \in \mathsf{LTS}(B)$, and $\mathcal{M} \in \mathsf{LTS}(A)$:*
1. *If $\mathcal{N}$ is in the subclass $\mathsf{LTS}(\mathsf{dom}(\mathcal{R})) \subseteq \mathsf{LTS}(B)$, then $\varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M}$ implies $\mathcal{N} \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{M})$.*
2. *If $\mathcal{M}$ is deterministic, then $\mathcal{N} \sqsubseteq \alpha_{\mathcal{R}}(\mathcal{M})$ implies $\varrho_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M}$.*

The condition in the first direction means that $\mathcal{N} \in \mathsf{LTS}(B)$ only makes use of action labels in the subset $\mathsf{dom}(\mathcal{R}) \subseteq B$. Hence, in the proof, we can consider $\mathcal{R}$ to be a total map $\mathsf{dom}(\mathcal{R}) \to A^+$.

▶ Remark 4.6. If we wanted to support non-deterministic $\mathcal{M}$, we can consider a less-pleasant $\varrho'_{\mathcal{R}}$ that replaces every $q \xrightarrow{b} q'$ for $\mathcal{R}(a_1 \cdots a_n) = b$ with literally a sequence $q \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'$. Thus, $\varrho'_{\mathcal{R}}$ would rather create a system as in Figure 1b whereas $\varrho_{\mathcal{R}}$ creates a system as in Figure 1c. However, such an operator $\varrho'_{\mathcal{R}}$ does not preserve determinism.

▶ Remark 4.7. In the proof of the Galois connection, we make use of the fact that our action codes are functional, i.e. that every $b \in B$ is encoded by at most one $w \in A^*$. We would allow multiple, then one can show that α can not have a left-adjoint (details in appendix).

In the first direction, we can even prove a stronger statement for $\mathcal{M} := \varrho_{\mathcal{R}}(\mathcal{N})$, showing a Galois insertion between $\alpha_{\mathcal{R}}$ and $\varrho_{\mathcal{R}}$:

▶ **Theorem 4.8** (Galois insertion,). *For $\mathcal{R} \in \mathsf{Code}(A, B)$, if $\mathcal{N} \in \mathsf{LTS}(B)$ is in the subclass $\mathcal{N} \in \mathsf{LTS}(\mathsf{dom}(\mathcal{R}))$, then $\mathcal{N} \cong \alpha_{\mathcal{R}}(\varrho_{\mathcal{R}}(\mathcal{N}))$.*

5 Concretizations

In this section, we consider another method of transforming an abstract system into a concrete one: the *concretization* operator. Whereas refinement is the left adjoint of contraction (Theorem 4.5), this section will establish that concretization is the right adjoint (Theorem 5.5) of contraction. Whereas for refinement we omitted transitions for which the action code $\mathcal{R}$ was not defined, for concretization we add transitions to a new *chaos* state [20] in which any action may occur. Essentially, this is the idea of *demonic completion* of [6]. In order to reduce the number of transitions to the chaos state, the concretization operator is parametric in a reflexive relation $\mathcal{I} \subseteq A \times A$ which describes whether two symbols are sufficiently similar. With this relation, we allow transitions to the chaos state only for those symbols that are not similar to any symbol for which the code is defined:

▶ **Definition 5.1** (Concretization,). *Let $\mathcal{M} \in \mathsf{LTS}(B)$ be an LTS, $\mathcal{R} \in \mathsf{Code}(A, B)$ an action code, and $\mathcal{I} \subseteq A \times A$ a reflexive relation. The* concretization *$\gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M}) \in \mathsf{LTS}(A)$ consists of:*

- $Q^{\gamma(\mathcal{M})} := Q^{\mathcal{M}} \times W \cup \{\chi\}$ *with* $W := \{w \in A^* \mid w = \varepsilon$ *or* $\exists b \in \mathsf{dom}(\mathcal{R})\colon w \lneqq \mathcal{R}(b)\}$.
- $q_0^{\gamma(\mathcal{M})} := (q_0^{\mathcal{M}}, \varepsilon)$
- *Transitions are defined by the following rules, for $a \in A$, $w \in A^*$, $b \in B$:*

$$\frac{wa \in W}{(q, w) \xrightarrow{a}_{\gamma(\mathcal{M})} (q, wa)}\ (1_\gamma) \qquad \frac{q \xrightarrow{b}_{\mathcal{M}} q', \quad \mathcal{R}(b) = wa}{(q, w) \xrightarrow{a}_{\gamma(\mathcal{M})} (q', \varepsilon)}\ (2_\gamma)$$

$$\frac{\forall a' \in A, (a, a') \in \mathcal{I}\colon \quad wa' \notin W \wedge wa' \notin \mathsf{im}(\mathcal{R})}{(q, w) \xrightarrow{a}_{\gamma(\mathcal{M})} \chi}\ (3_\gamma) \qquad \frac{}{\chi \xrightarrow{a}_{\gamma(\mathcal{M})} \chi}\ (4_\gamma)$$

Intuitively, W represents the internal nodes of the tree-representation of action code $\mathcal{R}$. The transitions then try to accumulate a word $w \in A^*$ known to the action code (rule (1_γ)). As soon as we reach $w = \mathcal{R}(b)$ for some b, we use a b-transition in the original $\mathcal{M} \in \mathsf{LTS}(B)$ to jump to a new state (rule (2_γ)). The chaos state χ attracts all runs with symbols unknown to the action code. The corresponding rule (3_γ) involves the relation $\mathcal{I} \subseteq A \times A$. The rule only allows a transition to χ for a symbol $a \in A$ if there is no related symbol $a' \in A$, $(a, a') \in A$ for which the code R could make a transition. For general LTSs, we can simply consider $\mathcal{I}$ to be the identity relation on A. Once transitioned to the chaos state χ, we allow transitions for arbitrary action symbols $a \in A$ (rule (4_γ)).

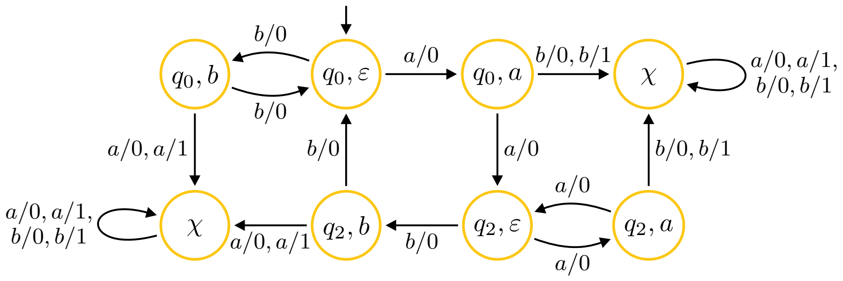


Figure 8 Concretization of the Mealy machine of Figure 5a.

▶ **Example 5.2.** For the special case of Mealy machines $A := I \times O$, we can define $\mathcal{I} \subseteq (I \times O) \times (I \times O)$ to relate (i, o) and (i', o') iff $i = i'$, i.e. two actions are related if they use the same input symbol. Then, we only have transitions to the chaos states if the code can't do any action for the same input symbol $i \in I$. Figure 8 depicts the concretization (for this $\mathcal{I}$) of the Mealy machine of Figure 5a(right) with the action code of Figure 5a(left). To increase readability, we introduced two copies of chaos state χ. Also, multiple labels next to an arrow denote multiple transitions.

Like in the refinement operator, the transition structure of γ is built in such a way that transitions for $b \in B$ in $\mathcal{M}$ correspond to runs of $\mathcal{R}(b)$ in $\gamma(\mathcal{M})$:

$$(q, \varepsilon) \xrightarrow{\mathcal{R}(b)}_{\gamma(\mathcal{M})} \bar{q} \quad \text{iff} \quad \exists q' : q \xrightarrow{b}_{\mathcal{M}} q' \text{ and } \bar{q} = (q', \varepsilon).$$

To make γ right adjoint to α, all runs outside the code $\mathcal{R}$ lead to the chaos state. One may think that the many transitions to the chaos state χ would make the construction $\gamma_{\mathcal{R}}$ trivial. However, only those paths lead to χ for which the action code is not defined.

The following technical condition describes that a code $\mathcal{R}$ contains sufficiently many related symbols compared to a given $\mathcal{M} \in \mathsf{LTS}(A)$:

▶ **Definition 5.3** (). *A code $\mathcal{R} \in \mathsf{Code}(A, B)$ is called $\mathcal{I}$-*complete for $\mathcal{M} \in \mathsf{LTS}(A)$*, if for all $w \in B^*$, $u \in A^*$, $q \in Q^{\mathcal{M}}$, $a, a' \in A$:*

$$r_0 \xrightarrow{u\,a}_{\mathcal{R}} \quad \text{and} \quad (a, a') \in \mathcal{I} \quad \text{and} \quad q_0 \xrightarrow{\mathcal{R}^*(w)\ u}_{\mathcal{M}} q \xrightarrow{a'} \quad \text{implies} \quad r_0 \xrightarrow{u\,a'}_{\mathcal{R}}.$$

Intuitively, $\mathcal{I}$-completeness means that if a state $q \in \mathcal{M}$ can do a transition for $a' \in A$ which is related to similar symbol $a \in A$ defined in the action code, then $a' \in A$ itself is also defined in the action code. However, we do not compare arbitrary transitions of q in $\mathcal{M}$ with arbitrary symbols mentioned in $\mathcal{R}$, but only look at the node in $\mathcal{R}$ reached when 'executing $\mathcal{R}$' zero or more times while following the path $q_0 \Longrightarrow q$.

For example, if $\mathcal{I} \subseteq A \times A$ happens to be the identity relation, then $\mathcal{R}$ is $\mathcal{I}$-complete for any $\mathcal{M} \in \mathsf{LTS}(A)$. In the instance of $\mathcal{I} \subseteq (I \times O) \times (I \times O)$ for Mealy machines, if $\mathcal{R}$ is $\mathcal{I}$-complete for $\mathcal{M}$, then this means: whenever a state $q \in Q^{\mathcal{M}}$ has transitions $q \xrightarrow{i/o}$ and $q \xrightarrow{i/o'}$, then the code $\mathcal{R}$ is defined for either both or none of them.

▶ **Assumption 5.4.** For the rest of the present Section 5, we fix the sets A, B, an action code $\mathcal{R} \in \mathsf{Code}(A, B)$, and a reflexive relation $\mathcal{I} \subseteq A \times A$.

▶ **Theorem 5.5** (Galois connection,). *For all $\mathcal{N} \in \mathsf{LTS}(A)$, and $\mathcal{M} \in \mathsf{LTS}(B)$, such that $\mathcal{R}$ is $\mathcal{I}$-complete for $\mathcal{N}$, we have*

$$\alpha_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \ (\text{in } \mathsf{LTS}(B)) \qquad \Longleftrightarrow \qquad \mathcal{N} \sqsubseteq \gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M}) \ (\text{in } \mathsf{LTS}(A)).$$

▶ **Example 5.6** (). If we instantiate $\mathcal{I}$ to be the identity relation Δ on A, then this means that we simply replace a' with a in rule (3_γ), and then we have above equivalence for all $\mathcal{N} \in \mathsf{LTS}(A)$ and $\mathcal{M} \in \mathsf{LTS}(B)$ (without any side-condition):

$$\alpha_{\mathcal{R}}(\mathcal{N}) \sqsubseteq \mathcal{M} \text{ (in } \mathsf{LTS}(B)) \qquad \Longleftrightarrow \qquad \mathcal{N} \sqsubseteq \gamma_{\mathcal{R},\Delta}(\mathcal{M}) \text{ (in } \mathsf{LTS}(A)).$$

▶ **Example 5.7** (). Consider the instantiation of $\mathcal{I}$ for Mealy machines described in Example 5.2. Let $\mathcal{N}$ be our running example of Figure 2, let $\mathcal{R}$ be the action code from Figure 5a(left), and let $\mathcal{M}$ be the abstract Mealy machine from Figure 5a(right), i.e. $\alpha_{\mathcal{R}}(\mathcal{N}) = \mathcal{M}$. One can verify that $\mathcal{R}$ is $\mathcal{I}$-complete for $\mathcal{N}$. Therefore, application of the Galois connection gives that there is a simulation from $\mathcal{N}$ to the Mealy machine $\gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M})$ of Figure 8.

It is a standard proof that the operators in a Galois connections are monotone. In that proof, one applies the Galois connection also to $\mathcal{M} := \gamma_{\mathcal{R},\mathcal{I}}(\mathcal{N})$, so we first need to show that it satisfies the technical completeness condition:

▶ **Lemma 5.8** (). *$\mathcal{R}$ is always $\mathcal{I}$-complete for $\gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M})$.*

▶ **Corollary 5.9** (). *$\mathcal{M} \sqsubseteq \mathcal{N}$ in $\mathsf{LTS}(B)$ implies $\gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M}) \sqsubseteq \gamma_{\mathcal{R},\mathcal{I}}(\mathcal{N})$ in $\mathsf{LTS}(A)$.*

▶ Remark 5.10. Monotonicity of concretization also follows by observing that the rules in Definition 5.1 all fit the *tyft* format of [19] if we view $(\cdot, w)$ as a unary operator for each sequence $w \in W$. Monotonicity then follows from the result of [19] that the simulation preorder is a congruence for any operator defined using the *tyft* format. Since contraction also can be defined using the *tyft* format, also monotonicity of contraction (Proposition 3.11) follows from the result of [19].

Like refinement, concretization preserves determinism.

▶ **Proposition 5.11** (). *If $\mathcal{M} \in \mathsf{LTS}(B)$ is a deterministic LTS and Δ the identity relation on A, then $\gamma_{\mathcal{R},\Delta}(\mathcal{M})$ is deterministic, too.*

If the code $\mathcal{R} \in \mathsf{Code}(A, B)$ is defined for all labels mentioned in $\mathcal{M} \in \mathsf{LTS}(B)$, then $\gamma_{\mathcal{R}}$ is even the right inverse of $\alpha_{\mathcal{R}}$, that is, we have a Galois insertion:

▶ **Theorem 5.12** (Galois insertion,). *If $\mathcal{M} \in \mathsf{LTS}(\mathsf{dom}(\mathcal{R}))$, then $\mathcal{M} \cong \alpha_{\mathcal{R}}(\gamma_{\mathcal{R},\mathcal{I}}(\mathcal{M}))$.*

Note that $\mathsf{dom}(\mathcal{R}) \subseteq B$, and so $\mathsf{LTS}(\mathsf{dom}(\mathcal{R})) \subseteq \mathsf{LTS}(B)$. Since we may reach the chaos state χ in the concretization, it is clear that $\gamma_{\mathcal{R}}$ is not a left inverse of $\alpha_{\mathcal{R}}$ in general.

6 Action Code Composition

Since notions of abstraction can be stacked up, it is natural to consider multiple adaptors for multiple action codes. Assume an action code $\mathcal{R} \in \mathsf{Code}(A, B)$ and an action code $\mathcal{S} \in \mathsf{Code}(B, C)$. Then the composition of $\mathcal{R}$ and $\mathcal{S}$ should be an action code from A to C.

▶ **Definition 6.1** (). *Given two map-based action codes $\mathcal{R}\colon B \rightharpoonup A^+$ and $\mathcal{S}\colon C \rightharpoonup B^+$, we define their* (Kleisli) composition *$(\mathcal{R} * \mathcal{S})\colon C \rightharpoonup A^+$ by*

$$(\mathcal{R} * \mathcal{S})(c) = \begin{cases} \mathcal{R}(b_1) \cdots \mathcal{R}(b_n) & \textit{if } \mathcal{S}(c) = b_1 \cdots b_n \textit{ with } \forall i\colon b_i \in \mathsf{dom}(\mathcal{R}) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The composed action code $\mathcal{R} * \mathcal{S}$ is only defined for $c \in C$ if $\mathcal{S}$ is defined for c and additionally $\mathcal{R}$ is defined for every letter $b_i \in B$ that appears in the word $\mathcal{S}(c) \in B^+$.

▶ Remark 6.2. The defined composition is an instance of *Kleisli composition* for a monad, which is a standard concept in functional programming and category theory.

▶ **Lemma 6.3** (). *Action codes are closed under composition.*
Concretely, given two map-based action codes $\mathcal{R}\colon B \rightharpoonup A^+$ and $\mathcal{S}\colon C \rightharpoonup B^+$, their Kleisli composition $(\mathcal{R} * \mathcal{S})\colon C \rightharpoonup A^+$ is again a prefix-free *partial map.*

Now that we can compose action codes, we can now investigate how the previously defined operators on LTSs behave for composed action codes:

▶ **Theorem 6.4** (,). *Contraction and refinement commute with action code composition: for action codes $\mathcal{R} \in \mathsf{Code}(A, B)$, $\mathcal{S} \in \mathsf{Code}(B, C)$,*

1. $\alpha_{\mathcal{R}*\mathcal{S}}(\mathcal{M}) = \alpha_{\mathcal{S}}(\alpha_{\mathcal{R}}(\mathcal{M}))$ *for all $\mathcal{M} \in \mathsf{LTS}(A)$.*
2. $\varrho_{\mathcal{R}*\mathcal{S}}(\mathcal{M}) = \varrho_{\mathcal{R}}(\varrho_{\mathcal{S}}(\mathcal{M}))$*, whenever* $\mathsf{im}(\mathcal{S}) \subseteq \mathsf{dom}(\mathcal{R})^+$ *and for all $\mathcal{M} \in \mathsf{LTS}(C)$.*

For the case of refinement, the additional assumption expresses that every word produced by $\mathcal{S}$ only contains letters $b \in B$ for which $\mathcal{R}$ is defined. The equations of Theorem 6.4 equivalently mean that the following diagrams commute:

$$\begin{tikzcd}
\mathsf{LTS}(A) \arrow[rr, "\alpha_{\mathcal{R}*\mathcal{S}}"] \arrow[dr, "\alpha_{\mathcal{R}}"'] & & \mathsf{LTS}(C) \\
& \mathsf{LTS}(B) \arrow[ur, "\alpha_{\mathcal{S}}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{LTS}(A) & & \mathsf{LTS}(C) \arrow[ll, "\varrho_{\mathcal{R}*\mathcal{S}}"'] \arrow[dl, "\varrho_{\mathcal{S}}"] \\
& \mathsf{LTS}(B) \arrow[ul, "\varrho_{\mathcal{R}}"] &
\end{tikzcd}$$

▶ Remark 6.5 (). Concretization does not commute with action code composition. The reason for that is that the rules (1_γ) and (2_γ) in $\gamma_{\mathcal{R}}(\gamma_{\mathcal{S}}(\mathcal{M}))$ would also be applied to transitions for the chaos state in $\gamma_{\mathcal{S}}(\mathcal{M}) \in \mathsf{LTS}(B)$ (see appendix for details).

7 Adaptors

In this section, we describe how action codes may be used for learning and testing of black-box systems. The general architecture is shown in Figure 9. On the right we see the *system under test (SUT)*, some piece of hardware/software whose behavior can be modeled by a Mealy machine $\mathcal{M}$ with inputs I and outputs O. On the left we see the *learner/tester*, an

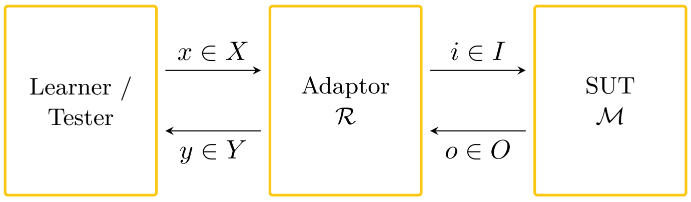


Figure 9 Using action codes for learning/testing.

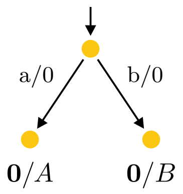


Figure 10 Example 7.4.

agent which either tries to construct a model $\mathcal{N}$ of $\mathcal{M}$ by performing experiments, or already has such a model $\mathcal{N}$ and performs experiments (tests) to find a counterexample which shows that $\mathcal{M}$ and $\mathcal{N}$ behave differently. The learner/tester uses abstract inputs X and outputs Y. In between the learner/tester and the SUT we place an *adaptor*, which uses action code $\mathcal{R}$ to translate between the abstract world of the learner/tester and the concrete world of the SUT. In order to enable the adaptor to do its job, we need to make four (reasonable) assumptions.

Our first assumption, common in model-based testing [35], is that the SUT will accept any input from I in any state, that is, we require that $\mathcal{M}$ is *input enabled*: for all $q \in Q^{\mathcal{M}}$ and $i \in I$, $q \xrightarrow{i/}_{\mathcal{M}}$. Our second assumption is that code $\mathcal{R}$ is $\mathcal{I}$-complete for $\mathcal{M}$ (for $\mathcal{I}$

denoting *same input* as in Example 5.2). This ensures that whenever the adaptor sends a concrete symbol $i \in I$ to the SUT, the adaptor will accept any output $o \in O$ that the SUT may possibly produce in response. Our third assumption is that whenever the adaptor receives an abstract input $x \in X$ from the learner/tester, it can choose concrete inputs from I that drive $\mathcal{R}$ from its initial state to a leaf with label (x, y), for some $y \in Y$. Output y can then be returned as a response to the learner/tester. Reaching such a leaf is nontrivial since the transitions taken in $\mathcal{R}$ are also determined by the outputs provided by the SUT. We may think of the situation in terms of a 2-player game where the adaptor wins if the game ends in an x-leaf, and the SUT wins otherwise. Formally, we require that $\mathcal{R}$ has finitely many states and a winning strategy for every input $x \in X$, as defined below:

▶ **Definition 7.1** (Winning). *Let $\mathcal{R} = \langle R, r_0, \longrightarrow, l\rangle \in \mathsf{Code}(I \times O, X \times Y)$ be an action code with R finite and let $x \in X$. Then*
1. *A leaf $r \in R$ is* winning for x *if $\pi_1(l(r)) = x$.*[1]
2. *An internal state $r \in R$ is* winning for x with input $i \in I$ *if $r \xrightarrow{i/}$ and, for each transition of the form $r \xrightarrow{i/o} r'$, r' is winning for x.*
3. *An internal state $r \in R$ is* winning for x *if it is winning for x with some $i \in I$.*
4. *$\mathcal{R}$ has a* winning strategy for x *if r_0 is winning for x.*

▶ **Example 7.2.** The action codes for Mealy machines that we have seen thus far (Figures 4, 5a and 5b) are winning for all the inputs that label their leaves. The action code of Figure 4 is not winning for the input (latte macchiato), for the simple reason that this input does not label any leaf. If we remove the transition to the leaf /2 in Figure 4, then the resulting code is no longer winning for (espresso), although it is winning for (coffee).

Our fourth and final assumption is that action code $\mathcal{R}$ is *determinate*. If an action code is determinate then, for each state r and abstract input x, there is at most one concrete input i such that r is winning for x with i.

▶ **Definition 7.3** (Determinate,). *An action code $\mathcal{R}$ is* determinate *if, for each state r, whenever $r \xrightarrow{i_1/} r_1$, $r \xrightarrow{i_2/} r_2$ and from both r_1 and r_2 there is a path to a leaf labeled with input x, then $i_1 = i_2$.*

▶ **Example 7.4.** All action codes for Mealy machines that we have seen thus far (Figures 4, 5a and 5b) are determinate. Figure 10 shows an action code that is not determinate: in the root two different concrete inputs a and b are enabled that lead to leaves with the same abstract input $\mathbf{0}$. Hence (trivially), this action code does have a winning strategy for input $\mathbf{0}$.

Algorithm 1 shows pseudocode for an adaptor that implements action code $\mathcal{R}$. During learning/testing, the adaptor records the current state of the action code in a variable r. When an abstract input x arrives, it first sets r to r_0. As long as current state r is internal, the adaptor chooses an input i that is winning for x, and forwards it to the SUT. When the SUT replies with an output o, the adaptor sets r to a state r' with $r \xrightarrow{i/o} r'$. When the new r is internal the adaptor chooses again a winning input, and updates its current state after interacting with the SUT, etc. When the new r is a leaf with label (x, y) then the adaptor returns symbol y to the learner/tester and waits for the next abstract input to arrive.

[1] We use projections functions π_1 and π_2 to denote the first and second element of a pair, respectively. So $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$.

Algorithm 1 Pseudocode for an adaptor that implements action code $\mathcal{R}$.

```
1:  while true do
2:      $x \leftarrow$ Receive-from-learner()
3:      $r \leftarrow r_0$
4:      while $r$ is internal do                          ▷ loop invariant: $r$ is winning for $x$
5:          $i \leftarrow$ unique input such that $r$ is winning for $x$ with $i$
6:          Send-to-SUT($i$)
7:          $o \leftarrow$ Receive-from-SUT()
8:          $r \leftarrow$ unique state $r'$ such that $r \xrightarrow{i/o} r'$        ▷ $\mathcal{R}$ is $\mathcal{I}$-complete for $\mathcal{M}$
9:      end while
10:     Send-to-learner($\pi_2(l(r))$)
11: end while
```

From the perspective of the learner/tester, the combination of the adaptor and SUT behaves the same as the contraction $\alpha_{\mathcal{R}}(\mathcal{M})$. In the appendix, we will formalize this statement by modeling both the combination of adaptor and SUT, as well as contraction $\alpha_{\mathcal{R}}(\mathcal{M})$ as expressions in the process calculus CCS [30], and then establish the existence of *delay simulations* between these expressions. This implies that both expressions have the same traces if we remove all occurrences of the synchronizations between adaptor and SUT, which are invisible from the perspective of the learner.

▶ **Theorem 7.5.** *Let $\mathcal{M} \in \mathsf{LTS}(I \times O)$ be an input enabled Mealy machine and let $\mathcal{R} \in \mathsf{Code}(I \times O, X \times Y)$ be a finite, determinate action code that has a winning strategy for every input in X and that is output enabled for $\mathcal{M}$. Then the composition of an implementation for $\mathcal{M}$ and an adaptor for $\mathcal{R}$ is delay simulation equivalent to an implementation for $\alpha_{\mathcal{R}}(\mathcal{M})$.*

▶ Remark 7.6. Requiring the existence of a determinate action code with a winning strategy for a Mealy machine is not a severe restriction. Definition 7.1 implicitly describes a bottom-up algorithm (linear in the size of the action code) that checks whether a winning strategy exists. Checking whether an action code is determinate is also easy. A sufficient (but not necessary) condition for an action code to be determinate and have a winning strategy is that when we project the action code to the inputs (with concrete inputs labeling the transitions and abstract inputs as label for the leaves) and merge isomorphic subtrees, then the result is still an action code (defined for all the abstract inputs). This is a natural condition that can also be used for the design of determinate action codes with a winning strategy: we start from an action code for the inputs and recursively add output labels starting from the root. Whenever, for a given input i, different outputs may occur, we make a copy of the subtree after i for each possible output o. Finally, the abstract outputs need to be defined in such a way that the labeling of the leaves remains injective.

Active automata learning algorithms and tools for Mealy machines typically assume that the system under learning is *output deterministic*[2]: the output and target state of a transition are uniquely determined by its source state and input.

▶ **Definition 7.7.** *Mealy machine $\mathcal{M}$ is* output deterministic *if, for each state q and input i,*

$$q \xrightarrow{i/o} r \wedge q \xrightarrow{i/o'} r' \quad \Rightarrow \quad o = o' \wedge r = r'.$$

[2] The notion of deterministic that we use in this article is the standard one for LTSs. In the literature on Mealy machines and FSMs, machines that we call output deterministic are called deterministic, and machines that we call deterministic are called observable.

For action codes that are determinate, contraction preserves output determinism. This property makes it possible to use existing automata learning tools to learn models of an output deterministic SUT composed with a determinate adaptor.

▶ **Proposition 7.8** (). *Suppose $\mathcal{M}$ is a Mealy machine and $\mathcal{R}$ is an action code. If $\mathcal{M}$ is output deterministic and $\mathcal{R}$ is determinate then $\alpha_{\mathcal{R}}(\mathcal{M})$ is output deterministic.*

8 Discussion and Future Work

Via the notion of action codes, we provided a new perspective on the fundamental question how high-level state machine models with abstract actions can be related to low-level models in which these actions are refined by sequences of concrete actions. This perspective may, for instance, help with the systematic design of adaptors during learning and testing, and the subsequent interpretation of obtained results. Our theory allows for action codes (such as in Figure 4) that are adaptive in the sense that outputs which occur in response to inputs at the concrete level may determine the sequence of concrete inputs that refines an abstract input. We are not aware of case studies in which such adaptive codes are used, but believe they may be of practical interest. One may, for instance, consider a scenario in which an abstract action AUTHENTICATE is refined by a protocol in which a user is either asked to authenticate by entering a PIN code, or by providing a fingerprint.

Close to our work are the results of Rensink and Gorrieri [31], who investigate vertical implementation relations to link models at conceptually different levels of abstraction. These relations are indexed by a refinement function that maps abstract actions into concrete processes. Within a setting of a CCS-like language, Rensink & Gorrieri [31] list a number of proof rules that should hold for any vertical implementation relation, and propose *vertical bisimulation* as a candidate vertical implementation relation for which these proof rules hold. In the setting of our paper, we can define two vertical implementation relations $\sqsubseteq^{\mathcal{R}}_{\gamma}$ and $\sqsubseteq^{\mathcal{R}}_{\varrho}$, for any action code $\mathcal{R}$, by

$$\mathcal{M} \sqsubseteq^{\mathcal{R}}_{\gamma} \mathcal{N} \Leftrightarrow \mathcal{M} \sqsubseteq \gamma_{\mathcal{R}}(\mathcal{N}) \quad \text{and} \quad \mathcal{M} \sqsubseteq^{\mathcal{R}}_{\varrho} \mathcal{N} \Leftrightarrow \mathcal{M} \sqsubseteq \varrho_{\mathcal{R}}(\mathcal{N}).$$

Then $\sqsubseteq^{\mathcal{R}}_{\varrho} \subseteq \sqsubseteq^{\mathcal{R}}_{\gamma}$ and both relations satisfy all language-independent proof rules of [31]. For instance, we have

$$\frac{\mathcal{M} \sqsubseteq \mathcal{M}' \quad \mathcal{M}' \sqsubseteq^{\mathcal{R}}_{\gamma} \mathcal{N}' \quad \mathcal{N}' \sqsubseteq \mathcal{N}}{\mathcal{M} \sqsubseteq^{\mathcal{R}}_{\gamma} \mathcal{N}}$$

(since $\gamma_{\mathcal{R}}$ is monotone and $\sqsubseteq$ is transitive). With the action code $\mathcal{R}$ of Figure 3, both implementation relations relate the LTSs of Figures 1c and 1a. However, the vertical bisimulation preorder of Rensink and Gorrieri [31] does not relate these LTSs, when using a code that maps a to $1\,4\,1$, and b to $1\,4\,2$. This suggests that bisimulations may not be suitable as vertical implementation relations.

Also close to our work are results of Burton et al. [8, 23], who propose a vertical implementation relation in the context of CSP. Instead of action codes, they use *extraction patterns*, a strict monotonic map $extr : Dom \to B^*$, where Dom is the prefix closure of a set $dom \subseteq A^*$ of concrete action sequences that may be regarded as complete. As a mapping from concrete to abstract sequences of actions, extraction patterns are more general than action codes. However, as extraction mappings are not required to have an inverse, establishing interesting Galois connections in this setting may be difficult. With an extraction pattern defined in the obvious way, the LTSs of Figures 1c and 1a are related by the implementation relation of [8]. We are not aware of any other vertical implementation relation proposed in the literature that handles our basic interface refinement example correctly. We find it surprising that the fundamental problem of refining inputs actions has not been properly addressed in the literature, except in some work that apparently has not been picked up outside Newcastle-upon-Tyne and Catania.

The action refinement operator $\varrho_{\mathcal{R}}$ that we study is similar to the one proposed by [16, 17]. It improves on the one from [16, 17] by not introducing unnecessary nondeterminism, as illustrated in the example of Figure 1. However, it falls short of the approach of [16, 17] by not considering concurrency. Another difference is that in [16, 17] $\mathcal{R}(b)$ can be an arbitrary system (including choice and parallel composition), whereas in our work it must be a sequence. But then [16, 17] did not have the dual contraction operator $\alpha_{\mathcal{R}}$. It would be very interesting to combine both approaches.

Our theory is orthogonal to the one of Aarts et al. [1], which explores the use of so-called *mappers* to formalize adaptors that abstract the large action alphabets of realistic applications into small sets of actions that can be handled by a learning tool. Aarts et al. [1] also describe the relation between abstract and concrete models using a Galois connection. In practical applications of model learning, it makes sense to construct an adaptor that combines a mapper in the sense of [1] with an action code as introduced in this paper. Fiterău-Broştean et al. [11] describe a small domain specific language to specify mapper components, and from which adaptor software can be generated automatically. It would be interesting to extend this domain specific language so that it may also be used to specify action codes.

We developed our theory for LTSs and Mealy machines, using the simulation preorder as the implementation relation. It would be interesting to transfer our results to other modeling frameworks, such as IOTSs [35] timed automata [3] and Markov Decision Processes, and to other preorders and equivalences in the linear-time branching-time spectrum for LTSs [15] and IOTSs [22]. An obvious direction for future work would be to explore how action codes interact with parallel composition. Here the work of [8, 23] may serve as a basis.

Different action codes lead to different contractions, and thereby to different abstract views of a system, see for instance Figures 5a and 5b. We may try to exploit this fact during learning and testing. For instance, if a system $\mathcal{M}$ is too big for state-of-the-art learning algorithms, we may still succeed to learn partial views using cleverly selected action codes. Using our Galois connections we then could obtain various upper and lower bounds for $\mathcal{M}$. Ideally, such an approach may even succeed to uniquely identify $\mathcal{M}$. In particular, learning algorithms such as $L^{\#}$ [38] that use observation trees as their primary data structure may exploit the use of different action codes, since the refinement operator $\varrho_{\mathcal{R}}$ and contraction operator $\alpha_{\mathcal{R}}$ transform observation trees for abstract actions into observation trees for concrete actions, and vice versa. Maarse [29] quantified the quality of a contraction $\alpha_{\mathcal{R}}(\mathcal{M})$ in terms of the graph-theoretic concept of *eccentricity*. If q and q' are states in an LTS $\mathcal{M}$ then $d(q, q')$ is defined as the number of transitions in the shortest path from q to q' (or ∞ if no such path exists). For any set of states $Q \subseteq Q_{\mathcal{M}}$, the *eccentricity* $\varepsilon(Q)$ is defined as $\max_{q' \in Q_{\mathcal{M}}} \min_{q \in Q} d(q, q')$, that is, the maximal distance one needs to travel to visit a state of $\mathcal{M}$, starting from a state of Q. A good contraction has a small set of states Q and a low eccentricity $\varepsilon(Q)$: it only covers a small subset Q of the states of $\mathcal{M}$, but any state from $\mathcal{M}$ can be reached via a few transitions from a Q-state.

References

1 F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015. doi:10.1007/s10703-014-0216-x.

2 M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 1(15):73–132, 1993.

3 G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA, pages 125–126. IEEE Computer Society, 2006.

4 J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra.* North-Holland, 2001.

5 J. Berstel and D. Perrin. *Theory of codes.* Academic Press, 1985.

6 M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003. doi:10.1007/978-3-540-24617-6_7.

7 M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement in conformance testing. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings*, volume 3502 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2005.

8 J. Burton, M. Koutny, and G. Pappalardo. Implementing communicating processes in the event of interface difference. In *2nd International Conference on Application of Concurrency to System Design (ACSD 2001), 25-30 June 2001, Newcastle upon Tyne, UK*, page 87. IEEE Computer Society, 2001. doi:10.1109/CSD.2001.981767.

9 G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT'14),* San Diego, California, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.

10 E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

11 P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In S. Chaudhuri and A. Farzan, editors, *Proceedings 28th International Conference on Computer Aided Verification (CAV'16),* Toronto, Ontario, Canada, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. doi:10.1007/978-3-319-41540-6_25.

12 P. Fiterău-Broştean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.

13 P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151, New York, NY, USA, 2017. ACM. doi:10.1145/3092282.3092289.

14 H. Garavel and F. Lang. Equivalence checking 40 years after: A review of bisimulation tools. In N. Jansen, M. Stoelinga, and P. van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning*, pages 213–265, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-15629-8_13.

15 R.J. van Glabbeek. The linear time – Branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.

16 R.J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science 1989, MFCS'89, Porabka-Kozubnik, Poland, August 28 - September 1, 1989, Proceedings*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 1989. `doi:10.1007/3-540-51486-4_71`.

17 R.J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001. `doi:10.1007/s002360000041`.

18 R. Gorrieri and A. Rensink. Action refinement. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. North-Holland, 2001.

19 J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

20 C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

21 J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

22 R. Janssen, F.W. Vaandrager, and J. Tretmans. Relating alternating relations for conformance and refinement. In W. Ahrendt and S. Lizeth Tapia Tarifa, editors, *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings*, volume 11918 of *Lecture Notes in Computer Science*, pages 246–264. Springer, 2019. `doi:10.1007/978-3-030-34968-4_14`.

23 M. Koutny and G. Pappalardo. The ERT model of fault-tolerant computing and its application to a formalisation of coordinated atomic actions. Report 636, Department of Computing Science, University of Newcastle upon Tyne, 1998. URL: `http://www.cs.ncl.ac.uk/publications/trs/papers/636.pdf`.

24 L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

25 D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

26 N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

27 N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

28 N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

29 T. Maarse. *Active Mealy Machine Learning Using Action Refinements.* Master's thesis, Radboud University, Institute for Computing and Information Sciences, August 2020.

30 R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

31 A. Rensink and R. Gorrieri. Vertical implementation. *Information and Computation*, 170(1):95–133, 2001. `doi:10.1006/inco.2001.2967`.

32 J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium*, pages 193–206, Washington, D.C., August 2015. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter`.

33 C. McMahon Stone, T. Chothia, and J. de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In J. López, J. Zhou, and M. Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 325–345. Springer, 2018. `doi:10.1007/978-3-319-99073-6_16`.

34 A.S. Tanenbaum and D. Wetherall. *Computer networks, 5th Edition.* Pearson, 2011. URL: `https://www.worldcat.org/oclc/698581231`.

35 J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17:103–120, 1996.

36 J. Tretmans. Model based testing with labelled transition systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

37 F.W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, February 2017. `doi:10.1145/2967606`.

38 F.W. Vaandrager, B. Garhewal, J. Rot, and T. Wißmann. A new approach for active automata learning based on apartness. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022. `doi:10.1007/978-3-030-99524-9_12`.

39 P. Verleg. *Inferring SSH state machines using protocol state fuzzing.* Master thesis, Radboud University Nijmegen, February 2016. URL: `https://www.ru.nl/publish/pages/769526/z07_patrick_verleg.pdf`.

40 M. Yannakakis. Hierarchical state machines. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, pages 315–330, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.