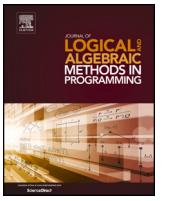
J. Log. Algebraic Methods Program. 148 (2026) 101092

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Combining sequential test cases into an equivalent set of adaptive test cases ☆

Robert M. Hierons [a], Mercedes G. Merayo [b], Manuel Núñez [b,*]

[a] School of Computer Science, University of Sheffield, S1 4DP, Sheffield, United Kingdom
[b] Design and Testing of Reliable Systems, Universidad Complutense de Madrid, 28040, Madrid, Spain

A B S T R A C T

When testing a state-based system one might use a set of (negative) test cases in which each test case is a sequence of events that should not occur. Testing then involves executing the system under test (SUT) in order to check whether any of these disallowed sequences can occur. While testing using such sequences can be effective, they introduce a source of inefficiency: if a test case expects the SUT to produce output a after observing a sequence σ and the SUT instead produces a different output a' after σ then testing with that test case did not show an error, because the SUT can autonomously produce outputs, and terminates because the test case only makes sense if the exact sequence is observed. This is a source of inefficiency if there is another test case that starts with σ followed by a': we could have continued evaluating whether the application of this second test case leads to an error. This paper considers scenarios in which events represent inputs, outputs, or the passing of discrete time. We show how a set of sequential test cases can be converted into an equivalent set of adaptive test cases, with adaptivity addressing the above source of inefficiency. The proposed approach has the potential to improve efficiency when using any test generation technique that returns negative sequential test cases.

1. Introduction

Testing is the process of evaluating a system according to how it reacts to certain external stimuli. The most common approach to testing consists of applying inputs to the system under test (SUT), observing the outputs that the SUT produces, and deciding whether the observation is the expected one or not. There are several difficulties involved in this apparently simple process. For example, we have to decide which inputs to apply, implement the observation of outputs and decide whether the observed outputs correspond to a faulty behaviour of the system.

The work reported in this paper is motivated by the problem of testing a state-based system and so systems behaviours are sequences of events. Most testing approaches execute the SUT with test cases in order to check whether the SUT has disallowed behaviours and the choice of approach used can depend on the types of events that can be observed. For example, we might check whether the SUT shows faulty behaviours with respect to the time that it takes to produce outputs (see [1–3] for recent work within the scope of this paper). This paper considers testing approaches in which observations (events) represent inputs, outputs, or the passing of discrete time. If there is a formal model that specifies the allowed behaviours then this model might be used as the basis for deriving test cases [4–6]. However, often there is no such formal model and so there is a need for alternative approaches.

In this paper we assume that we have a set of sequential test cases (STCs), where each of them is a sequence (*trace*) σ of events such that the SUT should *not* be able to perform σ. Such STCs might have been derived from the system requirements or from a (possibly

☆ Research partially supported by EPSRC project RoboTest: Systematic Model-Based Testing and Simulation of Mobile Autonomous Robots, EP/R025134/1, the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2021-122215NB-C31 (AwESOMe) and the Comunidad de Madrid under grant TEC-2024/COM-235 (DESAFíO-CM).

* Corresponding author.
E-mail addresses: R.Hierons@sheffield.ac.uk (R.M. Hierons), mgmerayo@fdi.ucm.es (M.G. Merayo), mn@sip.ucm.es (M. Núñez).

https://doi.org/10.1016/j.jlamp.2025.101092
Received 28 February 2025; Received in revised form 28 May 2025; Accepted 23 August 2025

Available online 2 September 2025

formal) model. We also assume that these test cases are minimal failures: if $\sigma.a$ is an STC then $\sigma.a$ is not an allowed behaviour (it is *disallowed*) but σ is an allowed behaviour. We have two main reasons for considering STCs of this form. First, the set of allowed traces of a system is prefix closed and so if σ is a disallowed trace then so are all extensions of σ: there is no point in testing further if we observe σ. Second, there are several types of automated test generation techniques that return such minimal failures. For example, it is possible to derive such sequences directly from the semantics of a specification [1,7,8]. There are also mutation-based [9] test generation approaches in which a system or model P is mutated to create a mutant M and then a model-checker is used to check whether M conforms to P. If M does not conform to P then the model-checker returns a behaviour ρ, such as a trace [10], that demonstrates this. Assuming a minimal such behaviour is produced, ρ is of the form described above; otherwise, one can use a prefix of ρ. Finally, there are approaches that execute the SUT until a failure is observed, such as a vehicle that is being controlled getting too close to another vehicle, and then stop test execution (see, for example, [11]). This results in failing traces being observed and these might be used again in, for example, regression testing.

The use of disallowed behaviours of the form $\sigma.a$ can form the basis of systematic testing. The tester will attempt to 'apply' σ and then check whether a can occur after this. The process of attempting to apply σ is iterative: if $\sigma = a_1 \dots a_n$, for events $a_1, \dots, a_n$, then the tester will first try to apply a_1, if this succeeds then the tester next tries to apply a_2, etc. When some a_i is an input $?x$, the tester simply provides this input to the SUT. However, if a_i is an output $!o_1$ or represents the passing of time then the situation is more complex: typically the tester cannot block an output and so an output $!o_2 \neq a_i$ might occur rather than a_i. If $!o_2$ is an allowed behaviour at this point then, normally, test execution stops. The tester then resets the SUT and again tests with $\sigma.a$, repeating this process a sufficient number of times to either observe a failure or apply a fairness assumption to deduce that $\sigma.a$ is not a behaviour of the SUT (i.e. make the assumption that all possible outcomes have been observed).

The above process, for testing based on some $\sigma.a$, is effective if a fairness assumption can be made but it need not be *efficient*. To see why, consider the case where $\sigma = \sigma_1.!o_1.\sigma_2$ for two traces σ_1, σ_2 and output $!o_1$. As previously mentioned, normally the tester cannot block an output and so a different output $!o_2$ might occur after σ_1: the tester then stops the application of the test case, resets the SUT, and starts test execution again. However, it is possible that there is another test case to be used that is of the form $\sigma_1.!o_2.\sigma_3.b$: after observing $\sigma_1.!o_2$ the tester might have followed this second test case rather than stopping test execution. If the tester were to choose to follow the second test case at this point, then the tester is essentially combining these two test cases to form a tree rather than a sequence: a tree (an *adaptive test case*) that has two edges (branches) after σ_1. The above shows that the use of such trees has the potential to improve test efficiency.

In this paper we explore the problem of constructing a set A of adaptive test cases (ATCs) that is equivalent, in terms of fault finding ability, to a set S of STCs. We consider test cases that are to be applied to systems in which traces can contain inputs, outputs and also the passing of discrete time (represented by a symbol $\ominus$). The proposed approach is iterative: it maintains a set A of ATCs and in each iteration it takes an STC σ and attempts to merge σ with an ATC $p \in A$ to form a new ATC p'. We formalise the requirements of a step by defining what it means for one ATC p' to be the *union* of p and σ. We base the approach on two types of results: results regarding conditions under which, given an ATC p and an STC σ, there is an ATC p' that is the union of p and σ; and results that show how such an ATC p' can be constructed. The correctness of the algorithm follows from these results.

We formulate the problem in terms of allowed and disallowed traces, rather than assuming that there is a specification that defines these, because we aimed for generality. An alternative would be, for example, to have used a Labelled Transition System (LTS) that defines the sets of allowed and disallowed traces. We initially considered this approach and it would have fitted with one motivation for our research: test derivation from a model written in RoboChart [12], a domain specific language for robotics whose semantics is defined by a mapping to a timed version of CSP. However, we wanted to have a more general setting because the problem is more general. As an example, there is research in the area of testing cyber-physical systems [13] where we do not have a formal specification in the form of an LTS and instead there is a set of properties that behaviours should satisfy (e.g. the volume of liquid should not exceed a bound, the temperature should remain within a range, a vehicle should not collide with other vehicles). One might be able to phrase such problems in terms of an implicit LTS but this seems rather artificial.

In terms of related work, as far as we know, this is the first paper to explore the problem of converting a set of sequential test cases into a set of adaptive test cases. Sequential test cases are returned by many test generation algorithms designed for testing state-based systems. For example, most work on testing from a finite state machine produces sequential test cases [14] and tools/techniques that use model-checkers in test generation typically produce sequential test cases. In particular, our interest in this problem arose from us using sequential test cases in robotics [2,3]. There is plenty of work on *adaptive testing*, that is, using test cases that do not only apply an input and receive an output, but where the behaviour of a test case depends on the outputs that have been received. In particular, we have recent work on the topic [15]. The adaptive test cases used in this paper are similar to those used with ioco and its variants [16]. However, they are fundamentally different in the way they are constructed: sound test cases are constructed in ioco from a specification, with test generation typically being random. In contrast, we construct adaptive test cases from a set of sequential test cases (sequences that should not be observed) that can be seen as "requirements".

Given that there are test generation approaches that produce adaptive test suites, a legitimate question is why do we start from a non-adaptive test suite. The problem is that many test generation techniques directly return sequences rather than trees. In particular, as previously mentioned, we were motivated by the following two types of test generation techniques: mutation-based test generation [17,18] and test generation techniques used with cyber-physical systems that test until a failure is observed [11,13,19].

The paper is structured as follows. We start, in Section 2, by defining STCs, ATCs, and the notation and terminology used throughout the paper. Section 3 then describes the observations that can be made when using an STC or ATC and use this to formalise the problem considered. In Section 4 we then explore the problem of determining whether an ATC and an STC can be combined and define the

notion of an ATC p' being the union of an ATC p and an STC σ. This is used in Section 5 to define an algorithm for converting a set of STCs into an equivalent set of ATCs. Finally, we draw conclusions and describe possible lines of future work in Section 6.

2. Preliminaries

Throughout this paper we will let Σ denote the (finite) set of basic events that can be observed. Then Σ can be partitioned into two disjoint sets: a set I of system inputs and a set O of system outputs. Often we will precede the name of an input with '?' (e.g. $?i$) and the name of an output with '!' (e.g. $!o$). We will allow the observation of discrete time and so we include a special event $\ominus$, called "tock", that corresponds to the passing of one unit of time. We use $\Sigma_\ominus$ to denote the set $I \cup O \cup \{\ominus\}$ of inputs, outputs and $\ominus$ events. The observations made during testing will be sequences of events, which we call *traces*. $\Sigma_\ominus^*$ denotes the set of finite, possibly empty, traces, while $\Sigma_\ominus^+$ denotes the set of non-empty traces. Given two traces $\sigma_1, \sigma_2 \in \Sigma_\ominus^*$, we denote by $\sigma_1.\sigma_2$ the concatenation of these traces. We use ϵ to denote the empty trace.

We assume that some traces are known to be *allowed* (they are behaviours deemed to be correct) and some traces are known to be *disallowed* (they are behaviours deemed to be incorrect). By a trace σ being *allowed* we typically mean that σ is a trace of a (possibly implicit) specification. Such a specification could be a formal model but it might also be a set of properties that the SUT should satisfy. Note, however, that the proposed technique for mapping a set of STCs to a set of ATCs will not require the specification. Indeed, the sets of allowed traces and disallowed traces could be derived from the STCs provided as input to the technique since these STCs are disallowed traces of the form $\sigma.a$ such that σ is allowed.

We also make the following assumptions regarding the SUT that are standard in many testing scenarios, such as testing based on ioco [16].

- The SUT is input-enabled: the SUT must be ready to accept any input provided by the environment.
- The SUT has urgent outputs: if the SUT is in a state where it can produce an output then it does not allow time to pass (i.e. the SUT produces an output or receives an input before time passes).
- The set of allowed traces is prefix closed.

In testing, the tester (or the corresponding test case) supplies inputs and the SUT produces outputs. Since the SUT is input-enabled, if the tester applies an input then the SUT must be ready to receive it. In addition, system outputs cannot be blocked by the environment/tester. As previously noted, these are standard assumptions in ioco-like [16] frameworks.

In this paper we consider two types of test cases: *sequential* and *adaptive* test cases (these are formally defined below in Definitions 1 and 4).

1. *Sequential test case* (STC). These are in the form of a trace $\sigma.a$ such that σ is an allowed trace and a is an output or tock that is not allowed after σ. At each point the tester either supplies an input or observes an output or the passage of time. If the trace $\sigma.a$ is observed then testing stops with verdict *fail*. If the tester has observed trace σ_1, is waiting for a and the SUT instead produces an output $!o \neq a$ then testing stops with verdict *ok*, the system is reset, and (if necessary) the test runs again.
2. *Adaptive test case* (ATC). In this case, the action of the tester (apply an input or observe an output or the passing of time) depends on the sequence of observations that has been made in the current test execution. Adaptive test cases can be represented as trees.

As previously mentioned, ATCs have benefits in terms of efficiency of test execution. The main problem that we consider in this paper is to take a set of fixed STCs and generate from this a set of ATCs with the same effectiveness. Next, we present the formal definitions of our types of test cases.

We start by defining STCs, which are the input to the approach described in this paper. As previously mentioned, STCs will have a particular form: they are *disallowed* traces of the form $\sigma.a$ such that σ is an allowed trace.

Definition 1 *(Sequential Test Cases).* A *sequential test case* (STC) is a trace $\sigma.a \in (I \cup O \cup \{\ominus\})^*$ such that σ is an allowed trace, $\sigma.a$ is a disallowed trace, and $a \in O \cup \{\ominus\}$.

We require that $\sigma.a$ is disallowed since otherwise there would be no point in using such a test case. We require that σ is allowed since if σ is disallowed then, since the set of allowed behaviours is prefix-closed, there is no point in testing further after observing σ (all extensions of σ must be disallowed). We require that $a \in O \cup \{\ominus\}$ since we assume that the SUT will not block input: if σ is allowed and $?i \in I$ then $\sigma.?i$ must also be allowed.

Next we introduce a timed version of input output labelled transition systems that will be used to represent ATCs.

Definition 2. An *Input Output Labelled Transition System with tock* (IO$\ominus$LTS) is defined by a tuple $p = (Q, q_0, I, O, T)$ where

- Q is a countable, non-empty set of states;
- $q_0 \in Q$ is the initial state;
- I is the finite set of inputs and O is the finite set of outputs;
- $T \subseteq Q \times \Sigma_\ominus \times Q$ is the transition relation.

The IO⊖LTS p is initially in state q_0. If p is in state $q \in Q$ and performs an event a such that $(q, a, q') \in T$, for some state $q' \in Q$, then the system can move to state q' through this event a. We say that a is the *label* of t and that a is *enabled* in q. During the rest of the paper we will use $p, p', \ldots$ to denote IO⊖LTSs and $q, q', q_0, \ldots$ to denote states of IO⊖LTSs.

Next we introduce notation used during the rest of the paper.

Definition 3. Let $p = (Q, q_0, I, O, T)$ be an IO⊖LTS, $q, q' \in Q$ be states of p, $A \subseteq \Sigma_\ominus$ be a set of events, $a, a_1, \ldots, a_n \in \Sigma_\ominus$, with $n \geq 1$, be events and $\sigma, \sigma' \in \Sigma_\ominus^*$ be sequences of events.

$$q \xrightarrow{a} q' \Leftrightarrow_{\text{def}} (q, a, q') \in T$$
$$q \xrightarrow{a} \Leftrightarrow_{\text{def}} \exists q' \in Q : (q, a, q') \in T$$
$$q \stackrel{\epsilon}{\Longrightarrow} q' \Leftrightarrow_{\text{def}} q = q'$$
$$q \stackrel{a_1 \ldots a_n}{\Longrightarrow} q' \Leftrightarrow_{\text{def}} \exists q_1, \ldots, q_{n-1} \in Q : q \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q'$$
$$q \stackrel{\sigma}{\Longrightarrow} \Leftrightarrow_{\text{def}} \exists q' \in Q : q \stackrel{\sigma}{\Longrightarrow} q'$$
$$q = after_p(\sigma) \Leftrightarrow_{\text{def}} q_0 \stackrel{\sigma}{\Longrightarrow} q$$
$$a = next_\sigma(\sigma') \Leftrightarrow_{\text{def}} \exists \sigma'' \in \Sigma_\ominus^* : \sigma = \sigma'.a.\sigma''$$
$$A = enabled(q) \Leftrightarrow_{\text{def}} \forall a \in \Sigma_\ominus : q \xrightarrow{a} \iff a \in A$$

We will use $after_p(\sigma)$ only in the context of ATCs. Since they will be, by construction, deterministic we have that $after_p(\sigma)$ is well defined.

An ATC is an IO⊖LTS, with some restrictions, where states are labelled with *verdicts*: *ok* and *fail*. The application of an ATC to an SUT will be *ok* or *fail* depending on the verdict of the last reached state of the ATC.

The verdict *ok* captures two types of verdict that are sometimes used: pass, which states that the SUT has passed the test case, and inconclusive, which states that no failure has been observed but the test objective has not been achieved (e.g. trying to trigger event a after trace σ). We use the single verdict *ok* in order to avoid a complication that can occur when combining STCs to form an ATC. Specifically, this is the situation in which there is a state of the ATC that corresponds to verdict pass in one test case and verdict inconclusive in the other test case. Under future work we discuss alternative approaches to allocating verdicts; it would be straightforward to adapt the approach given to such alternative verdicts.

The following defines the notion of an ATC.

Definition 4 *(Adaptive Test Cases).* An *Adaptive Test Case* (ATC) is a tuple (Q, q_0, I, O, T, V), where (Q, q_0, I, O, T) is an IO⊖LTS and $V : Q \longrightarrow \{ok, fail\}$ is a *verdict function*. In addition, the following conditions hold:

1. The state set Q is finite and the graph induced by p' is acyclic.
2. The ATC is deterministic: given states $q, q' \in Q$ and $a \in \Sigma_\ominus$, if $q \xrightarrow{a} q'$ then there is no state $q'' \neq q'$ such that $q \xrightarrow{a} q''$.
3. Outputs are always available, except for in leaves (which represent testing having finished): for all $q \in Q$, if $enabled(q) \neq \emptyset$ then $O \subseteq enabled(q)$.
4. At each state, at most one input can be applied: for all $q \in Q$ and inputs $?i_1, ?i_2 \in I$, if $\{?i_1, ?i_2\} \subseteq enabled(q)$ then $?i_1 = ?i_2$.
5. There is no state $q \in Q$ and input $?i \in I$ such that $\{?i, \ominus\} \subseteq enabled(q)$.

Since an ATC is an IO⊖LTS, we will use the notation introduced in Definition 3. Now consider the conditions given above. The first condition ensures that testing terminates since the test case has no infinite paths. The second condition simplifies definitions and is included since a test case should not be non-deterministic. In particular, it would make no sense to have the potential to apply a test case twice to a deterministic SUT, observe the same sequence of events in both applications, and have one of them returns an *ok* verdict while the other one returns a *fail* verdict. The last three conditions are also standard properties of test cases. First, unless a test case has terminated, it must always be able to observe any output that can be produced by the SUT since the environment cannot block output. Second, at any point a test case may decide to apply an input but this input must be unique; this again avoids a form of non-determinism (where the test case can non-deterministically choose to send either input $?i_1$ or input $?i_2$ to the SUT). Third, if the test case can apply an input then there is no ⊖ transition. This final condition is included because the SUT cannot block input (if an input is supplied then the SUT receives this input immediately). Let us remark that, again, these conditions are compatible with standard approaches such as the one used in ioco [16].

Note that a non-leaf state q of an ATC might have no inputs enabled and also not have ⊖ enabled. If q is reached in testing and the SUT is not able to produce an output then testing terminates. The returned verdict is thus $V(q)$. Termination (deadlock) occurs when the SUT cannot engage in any of the events belonging to $enabled(q)$ and so $V(q)$ might not be *fail*.

Example 1. Consider $I = \{?i_1, ?i_2\}$ and $O = \{!o_1, !o_2\}$. Fig. 1 presents three ATCs. A *start* label indicates initial states. For simplicity, we simply write f instead of *fail*.

In the first ATC (the left-most), the tester first attempts to supply input $?i_1$ but can also observe output. If output is initially observed then testing terminates with verdict *ok*. Otherwise, if $?i_1$ is sent to the SUT then either an output is produced next (and

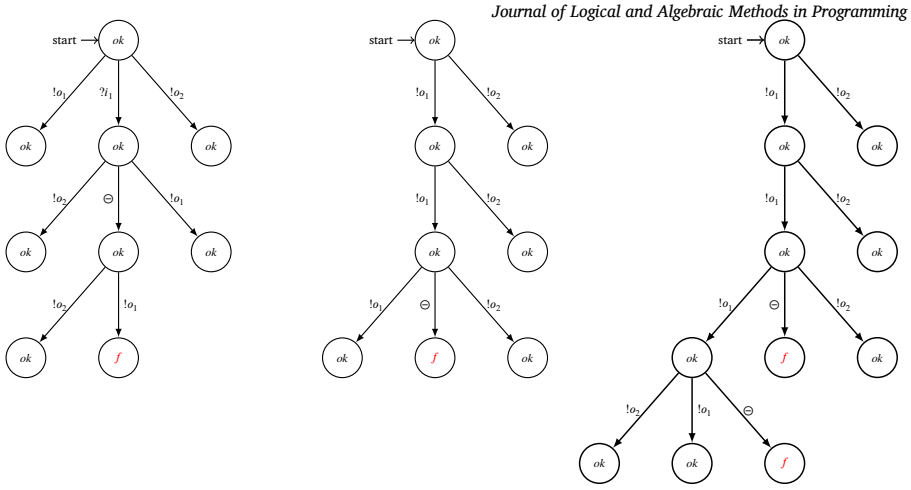


Fig. 1. Three ATCs.

testing terminates with verdict *ok*) or time passes. Finally, if $?i_1.\ominus$ has been observed, testing terminates with verdict *ok* if output $!o_2$ is produced and terminates with verdict *fail* if output $!o_1$ is produced. In addition, if testing terminates after $?i_1.\ominus$ has been observed then the verdict is *ok*. Thus, we can see that the SUT fails this test case if and only if test execution leads to the trace $?i_1.\ominus.!o_1$. The other two ATCs operate in a similar way. The SUT fails the second ATC if and only if test execution leads to the trace $!o_1.!o_1.\ominus$. In contrast, in the third ATC there are two traces that lead to verdict *fail*: $!o_1.!o_1.\ominus$ and $!o_1.!o_1.!o_1.\ominus$.

3. Possible observations: sequential and adaptive test cases

In this section we start by defining the set of possible observations that can be made when testing using an ATC and also when using an STC. This then allows us to define the set of possible observations that can be made when using a given test suite and makes it possible to formally define the problem that we consider in Section 4: that of converting a set of STCs into an 'equivalent' set of ATCs.

3.1. Possible observations of adaptive test cases

The set of possible observations of an ATC is the set of traces of that ATC.

Definition 5 *(Evolutions of an ATC).* Let $p = (Q, q_0, I, O, T, V)$ be an ATC. We define the set of evolutions of p, denoted by $Ev(p)$, as the following set of traces:

$$Ev(p) = \{\sigma \in \Sigma_{\ominus}^* \mid q_0 \stackrel{\sigma}{\Longrightarrow}\}$$

The set of evolutions of an ATC p defines exactly the traces that can be observed when testing with p.

The definition of an evolution does not take into account the verdicts of the states of the ATC. In particular, an evolution σ of an ATC can only find a fault if σ reaches a state of the ATC that has verdict *fail*.

Definition 6 *(Failing Evolutions of an ATC).* Let $p = (Q, q_0, I, O, T, V)$ be an ATC. We define the set of failing evolutions of p, denoted by $Ev_F(p)$, as the following set of traces:

$$Ev_F(p) = \{\sigma \in \Sigma_{\ominus}^* \mid \exists q \in Q : q_0 \stackrel{\sigma}{\Longrightarrow} q \wedge V(q) = fail\}$$

Recall that, by Definition 1, only particular traces can be used as STCs. Specifically, if $\sigma.a$ is an STC then σ is an allowed trace and $\sigma.a$ is a disallowed trace. We will also want to place restriction on ATCs, which we do through two concepts: an ATC being sound and an ATC being non-redundant. We start by saying what we mean by an ATC being sound.

Definition 7. An ATC p is sound if for every failing evolution $\sigma \in Ev_F(p)$ we have that σ is a disallowed trace.

Later we will prove that the way in which we construct ATCs (from STCs) ensures that these ATCs are sound (Proposition 5 in Section 4) and so there is no need to have access to the specification.

We also say what it means for an ATC to be non-redundant. Similar to above, we will prove that the ATCs we construct are non-redundant (Proposition 6 in Section 4).

Definition 8. An ATC p is *non-redundant* if it satisfies the following conditions.

1. If $\sigma \in Ev(p)$ is a disallowed trace then the state $after_p(\sigma)$ is a leaf.
2. If $\sigma \in Ev(p)$ is such that $q_1 = after_p(\sigma)$ is not a leaf and all states of p reached by transitions from q_1 are leaves then the following conditions hold.
   (a) There exists $q_2 \in Q$ and $a \in \Sigma_\ominus$ such that $q_1 \xrightarrow{a} q_2$ and $V(q_2) = fail$.
   (b) $enabled(q_1) \cap I = \emptyset$; and
   (c) If $q_1 \xrightarrow{\ominus} q_2$ then $V(q_2) = fail$.

The first condition simply says that if a disallowed trace σ is observed then testing stops. It does not, however, say what the verdict should be. The first part of the second condition ensures that if σ is a trace that reaches node q_1 of p that is not a leaf then further execution of p could potentially lead to verdict *fail* (Proposition 1 below); if this was not the case then there is no value in testing further and so q_1 should be a leaf.

The second part of the second condition is included since there is no value in a test case applying an input that takes p to a leaf since inputs are always enabled and so such an input cannot lead to verdict fail (i.e. applying such an input would be a waste). For the third part of the second condition, consider what happens if $\ominus \in enabled(q_1)$. We know, from the definition of an ATC, that no inputs are enabled at q_1 and all outputs are enabled at q_1. Further, if testing an SUT reaches state q_1 of p and the SUT can produce an output then such an output occurs and the transition from q_1 with label $\ominus$ is not taken (outputs are urgent). Thus, the transition t from q_1 with label $\ominus$ can only be followed if testing reaches q_1 and the SUT cannot produce an output at this point. However, if t takes p to a state with verdict *ok* then, since the verdict of q_1 is also *ok*, following the transition t does not change the verdict and so we can remove t without changing the effectiveness of the test case.

Proposition 1 below helps explain why Definition 8 is used. The result says that if a non-redundant ATC specifies that the tester should take some action in a state q_1 of an ATC then it is possible to reach a state with *fail* verdict from q_1. We want this property since otherwise testing should stop if the ATC reaches state q_1. We first prove a lemma.

Lemma 1. *Let $p = (Q, q_0, I, O, T, V)$ be a non-redundant ATC. If $q_1 \in Q$ is not a leaf then there is a trace σ and a state $q_2 \in Q$ such that $q_1 \stackrel{\sigma}{\Longrightarrow} q_2$ and $V(q_2) = fail$.*

Proof. We use proof by induction on the length ℓ of the longest path from q_1 to a leaf.

First consider the base case $\ell = 1$. In this case q_1 is not a leaf but all states of p reached from q_1 are leaves. Thus, by Definition 8 (second rule, first part) there must be at least one transition t from q_1 that takes p to a state q_2 such that $V(q_2) = fail$. Let us suppose that t has event b. The base case then follows with $\sigma = b$.

We now consider the inductive case. The inductive hypothesis is that the result holds if the length of the longest path from q_1 to a leaf is at most $\ell \geq 1$. We assume that the length of the longest path from q_1 to a leaf is $\ell + 1$. Choose some event $a \in enabled(q_1)$ that does not take p from q_1 to a leaf. Let q_3 be the state of p reached from q_1 by a (i.e. $q_1 \stackrel{a}{\Longrightarrow} q_3$). By the definition of q_3, we have that q_3 is not a leaf. In addition, since the length of the longest path from q_1 to a leaf is $\ell + 1$, the length of the longest path from q_3 to a leaf is at most ℓ. By the inductive hypothesis, we therefore have that there is a trace σ_1 and a node q_2 of p such that $q_3 \stackrel{\sigma_1}{\Longrightarrow} q_2$ and $V(q_2) = fail$. The result now follows by setting $\sigma = a.\sigma_1$. □

Proposition 1. *Let $p = (Q, q_0, I, O, T, V)$ be a non-redundant ATC and $\sigma_1.a \in Ev(p)$. Then, there exists a trace σ_2 such that $\sigma_1.\sigma_2 \in Ev_F(p)$.*

Proof. Let $q_1 = after_p(\sigma_1)$. (i.e. $q_0 \stackrel{\sigma_1}{\Longrightarrow} q_1$). Since $\sigma_1.a \in Ev(p)$, we have that q_1 is not a leaf. Thus, by Lemma 1, there is a trace σ_2 and a state q_2 of p such that $q_1 \stackrel{\sigma_2}{\Longrightarrow} q_2$ and $V(q_2) = fail$. Since $q_0 \stackrel{\sigma_1}{\Longrightarrow} q_1$ and $q_1 \stackrel{\sigma_2}{\Longrightarrow} q_2$ we have that $q_0 \stackrel{\sigma_1.\sigma_2}{\Longrightarrow} q_2$. Since $V(q_2) = fail$ we therefore have that $\sigma_1.\sigma_2 \in Ev_F(p)$ as required. □

3.2. Possible observations of sequential test cases

We want to compare ATCs and STCs and so also need to say what can be observed when using an STC σ. There are two reasons why this is not just the trace σ. First, in order to be consistent with ATCs we should include prefixes. Second, if $\sigma = \sigma_1.a.\sigma_2$ for some $\sigma_1, \sigma_2 \in \Sigma_\ominus^*$ and $a \in \Sigma_\ominus$ then, since a tester cannot block output, for all $b \in O \setminus \{a\}$ we have that testing with σ can also lead to the observation of $\sigma_1.b$. Given a trace σ, we use $pref(\sigma)$ to denote the set of prefixes of σ.

Definition 9 *(Evolutions of an STC).* Let $\sigma \in \Sigma_\ominus^+$ be an STC. We define the set of evolutions of σ, denoted by $Ev(\sigma)$, as the following set of traces:

$$Ev(\sigma) = pref(\sigma) \cup \{\sigma_1.b \mid \exists \sigma_2 \in \Sigma_\ominus^*, a \in \Sigma_\ominus : \sigma = \sigma_1.a.\sigma_2 \ \wedge \ b \in O \setminus \{a\}\}$$

Note that in the above definition, a might be an input. This would reflect the situation where the tester aims to apply an input but the SUT (autonomously) performs an output before the input is supplied.

It is straightforward to define the set of failing evolutions of an STC σ: it is just σ.

Definition 10 *(Failing evolutions of an STC).* Let $\sigma \in \Sigma^+_{\ominus}$ be an STC. We define the set of failing evolutions of σ by

$$Ev_F(\sigma) = \{\sigma\}$$

An alternative way of defining the evolutions of an STC σ would be to form an ATC from σ that essentially 'completes' σ. We now define such a completion. This is used in the algorithm given in Section 5.3 to allow us to add an ATC when we have an STC that cannot be merged with any of the current ATCs. In the following we assume that some arbitrary numbering of the outputs has been given and so $O = \{!o_1, \ldots, !o_\ell\}$ for some $\ell > 0$.

Definition 11. Given an STC $\sigma = a_1 \ldots a_k \in \Sigma^+_{\ominus}$, its completion $C(\sigma)$ is the ATC $p = (Q, q_0, I, O, T, V)$ in which we have the following.

1. $Q = \{q_0, \ldots, q_k\} \cup \{q^j_{i+1} \mid 0 \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
2. $T = \{(q_i, a_{i+1}, q_{i+1}) \mid 0 \leq i < k\} \cup \{(q_i, !o_j, q^j_{i+1}) \mid 0 \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
3. The verdict function is defined as follows

$$V(q) = \begin{cases} fail & \text{if } q = q_k \\ ok & \text{otherwise} \end{cases}$$

In the above definition, states in the set $\{q^j_{i+1} \mid 0 \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$ allow the observation of unanticipated output ($!o_j \neq a_{i+1}$) after a prefix $a_1 \ldots a_i$ of σ.

A *fail* verdict is assigned to the state of the ATC reached at the end of the sequence defining the corresponding STC and this is because, by the definition of an STC σ, we must have that σ is disallowed. The rest of the states are assigned verdict ok because either the application of the corresponding sequence was not complete (the test case application stopped with a prefix of σ) or the last event observed a was an output $!o$ produced by the SUT at a point where $!o$ is not the corresponding event in σ.

Example 2. Consider $I = \{?i_1, ?i_2\}$ and $O = \{!o_1, !o_2\}$. Fig. 1 (left) provides a graphical representation of the ATC $C(?i_1 . \ominus . !o_1)$ and Fig. 1 (centre) provides a graphical representation of the ATC $C(!o_1 . !o_1 . \ominus)$. If we consider the ATC in Fig. 1 (left), we see that the path to the state with verdict *fail* has the expected label $(?i_1 . \ominus . !o_1)$. Transitions have been added in order to ensure that all outputs are enabled on all (non-leaf) states on this path and all other verdicts are ok.

By construction, it is trivial to prove that the evolutions of an STC coincide with the evolutions of its completion.

Proposition 2. *Given $\sigma \in \Sigma^+_{\ominus}$, we have that*

- $Ev(\sigma) = Ev(C(\sigma))$*; and*
- $Ev_F(\sigma) = Ev_F(C(\sigma))$.

If addition, if σ is a trace of an ATC p then all traces of $C(\sigma)$ are also traces of p. The following is immediate from the definitions of an ATC and of $C(\sigma)$.

Proposition 3. *Given ATC p and trace σ, if $\sigma \in Ev(p)$ then $Ev(C(\sigma)) \subseteq Ev(p)$.*

3.3. Comparing test cases and test suites

Now that we have formally defined the set of observations that can be made when using a (sequential or adaptive) test case, we can formally compare test cases in terms of the faults that they can find. Recall that if σ is an STC then the set of possible responses of the SUT is the set of evolutions of the completion of σ: $Ev(C(\sigma))$. From this we can see that if S is a set of STCs then S might contain redundancy. The simplest form of redundancy is that which occurs when there are two STCs σ_1 and σ_2 such that $\sigma_1 = \sigma.a_1$ and $\sigma_2 = \sigma.a_2$ for some trace σ and $a_1 \neq a_2$. Here, the STCs σ_1 and σ_2 lead to the same possible observations even though neither is a prefix of the other.

Definition 12 *(Trace equivalence of Sequential Test Cases).* Let σ_1 and σ_2 be two STCs. We say that σ_1 and σ_2 are *trace equivalent* if $Ev(\sigma_1) = Ev(\sigma_2)$.

However, this says nothing about the verdicts. We can therefore define the following.

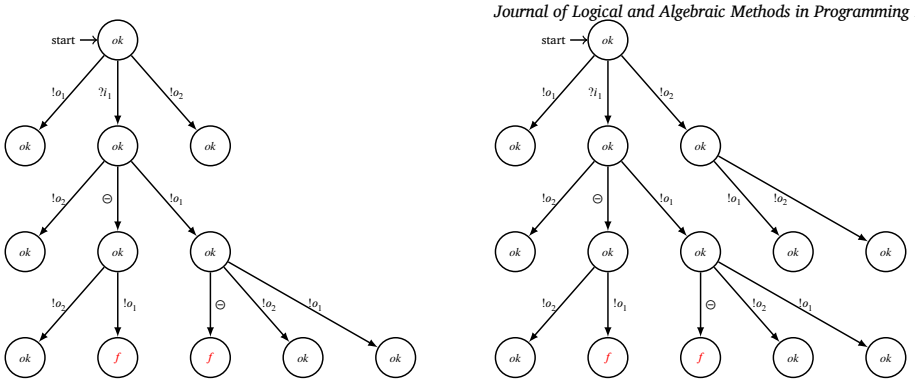


Fig. 2. An ATC that is the union of the ATC in Fig. 1 (left) and the STC $\sigma = ?i_1 !o_1 \ominus$ and a similar ATC (right) that does not represent such a union.

Definition 13 *(Equivalence of Sequential Test Cases).* Let σ_1 and σ_2 be two STCs. We say that σ_1 and σ_2 are *equivalent* if $Ev_F(\sigma_1) = Ev_F(\sigma_2)$.

Note that, from the above, we have that two STCs σ_1 and σ_2 are equivalent if and only if $\sigma_1 = \sigma_2$.

Proposition 4. *Given two STCs σ_1 and σ_2, σ_1 is equivalent to σ_2 if and only if $\sigma_1 = \sigma_2$.*

Finally, we can say what it means for a set A of ATCs to be equivalent, in terms of failures that can be observed, to a set S of STCs.

Definition 14 *(Equivalence).* A set A of ATCs is said to be *equivalent* to a set $S \subseteq \Sigma^+_\ominus$ of STCs if the following condition holds.

$$\bigcup_{\sigma \in S} Ev_F(C(\sigma)) = \bigcup_{p \in A} Ev_F(p)$$

4. Conditions under which we can combine STCs

In the previous section we provided a framework for reasoning about sequential and adaptive test cases. In Section 4.1 we formalise what we wish to achieve when we merge an STC σ and a sound and non-redundant ATC p. We then consider the conditions under which an STC σ can be removed without changing p (Section 4.2). Finally, we give conditions under which σ can be merged with an ATC p to form a new ATC p' (Section 4.3). The next section uses these when defining the proposed algorithm.

4.1. Objective

The proposed approach will be iterative, with each iteration aiming to combine an ATC p and an STC σ. In combining these, we want to construct an ATC p' such that p' finds exactly the same set of faulty behaviours found by separately testing with p and σ. We also want to produce an ATC p' that is non-redundant. The following formalises these requirements.

Definition 15. Given an STC σ and a sound and non-redundant ATC p, an ATC p' is said to be *the union* of p and σ if $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$ and $Ev(p') = Ev(p) \cup Ev(\sigma)$.

The first part of this definition ($Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$) ensures that p' finds the same faults as separately using p and σ. The second part ($Ev(p') = Ev(p) \cup Ev(\sigma)$) is included in order to avoid introducing additional traces that could lead to redundancy. The following example demonstrates this point.

Example 3. Consider the ATC p represented in Fig. 1 (left) and the STC $\sigma = ?i_1 !o_1 \ominus$. Fig. 2 (left) depicts the ATC p' corresponding to the union of both of them.

Now consider what could happen if we removed from the definition of union (Definition 15) the requirement that $Ev(p') = Ev(p) \cup Ev(\sigma)$. We can then, for example, change the ATC p' shown in Fig. 2 (left) to form a new ATC p'' by adding new states with verdict ok and transitions to these states from the state q_1 of p' such that $q'_0 \xrightarrow{!o_2} q_1$, where q'_0 is the initial state of p'. A possible such p'' is shown in Fig. 2 (right). Such a change preserves the set of failing evolutions and so we have that $Ev_F(p'') = Ev_F(p) \cup Ev_F(\sigma)$. However, $Ev(p'') \neq Ev(p) \cup Ev(\sigma)$ and so p'' is not the union of p and σ. We do not want to allow p'' to be the union of p and σ since p'' contains redundancy (failing the conditions of Definition 8). In practical terms, the states and transitions added to p' when

forming p'' could lead to an increase in testing effort, since testing no longer terminates if $!o_2$ is initially produced, but does not increase the ability of testing to find faults (since $Ev_F(p'') = Ev_F(p')$).

The proposed algorithm will combine/merge STCs with available ATCs one at a time. When we introduce the rules for merging an ATC and an STC, we will prove that each possible step (the application of a rule) involves replacing some p and σ with an ATC p' such that p' is the union of p and σ. We now prove that taking the union preserves both soundness and non-redundancy. We start by considering soundness.

Proposition 5. *Given an STC σ and a sound ATC p, if p' is the union of p and σ then p' is sound.*

Proof. By Definition 7, we are required to prove that all traces in $Ev_F(p')$ are disallowed. Since p' is the union of p and σ, by Definition 15 we have that $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$. We will separately consider the traces in $Ev_F(p)$ and $Ev_F(\sigma)$.

First consider the traces in $Ev_F(\sigma)$. By Definition 10 we have that $Ev_F(\sigma) = \sigma$ and by the definition of an STC (Definition 1) we know that σ is a disallowed trace. As a result, all traces in $Ev_F(\sigma)$ are disallowed.

Now consider the set $Ev_F(p)$ of traces. Observe that p is sound. Thus, by Definition 7, all traces in $Ev_F(p)$ are disallowed.

We now know that all traces in $Ev_F(\sigma)$ are disallowed and all traces in $Ev_F(p)$ are disallowed. Thus, all traces in $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$ are disallowed. By Definition 7, we therefore have that p' is sound as required. □

As previously mentioned, we wish to restrict attention to non-redundant ATCs and the following shows that taking the union with an STC preserves this property.

Proposition 6. *If $p = (Q, q_0, I, O, T, V)$ is a non-redundant ATC, σ is an STC, and $p' = (Q', q'_0, I, O, T', V')$ is the union of p and σ then p' is non-redundant.*

Proof. We will separately consider the conditions required for an ATC to be non-redundant.

Condition 1: We require that if $\sigma_1 \in Ev(p')$ is a disallowed trace then the state $after_{p'}(\sigma_1)$ is a leaf. We will use proof by contradiction and assume that this state is not a leaf. There is therefore some $a \in \Sigma_\ominus$ such that $\sigma_1.a \in Ev(p')$. Since $\sigma_1.a \in Ev(p')$ and p' is the union of p and σ we have that either $\sigma_1.a \in Ev(p)$ or $\sigma_1.a \in Ev(\sigma)$. Since σ_1 is a disallowed behaviour, the case where $\sigma_1.a \in Ev(p)$ contradicts p being non-redundant. Further, since σ_1 is a disallowed behaviour, the case where $\sigma_1.a \in Ev(\sigma)$ contradicts the definition of an STC. Since both cases lead to a contradiction, the result follows.

Condition 2(a): We are required to prove that if $\sigma_1 \in Ev(p')$ is such that $q^0_{p'} = after_{p'}(\sigma_1)$ is not a leaf and all states of p' reached by transitions from this state are leaves then at least one of them has verdict *fail*. We have that since $q^0_{p'}$ is not a leaf of p' there is some $a \in \Sigma_\ominus$ such that $\sigma_1.a \in Ev(p')$ and so either $\sigma_1.a \in Ev(p)$ or $\sigma_1.a \in Ev(\sigma)$.

First, consider the case where $\sigma_1.a \in Ev(p)$ and so the state $q^0_p = after_p(\sigma_1)$ is not a leaf. Since ATCs are deterministic, $Ev(p) \subseteq Ev(p')$, and all states of p' reached by transitions from $q^0_{p'}$ are leaves, we must have that all states of p reached by transitions from q^0_p are also leaves. But, since p is non-redundant, this means that some state of p reached by a transition from q^0_p has verdict *fail*. Since $Ev_F(p) \subseteq Ev_F(p')$ and p' is deterministic, we must therefore have that a state of p' reached by a transition from $q^0_{p'}$ has verdict *fail* as required.

Now consider the case where $\sigma_1.a \notin Ev(p)$ and so $\sigma_1.a \in Ev(\sigma)$. Since all states of p' reached by transitions from $q^0_{p'}$ are leaves, we must have that $\sigma = \sigma_1.a$. But, this means that $\sigma_1.a \in Ev_F(\sigma)$ and so, since p' is the union of p and σ, $\sigma_1.a \in Ev_F(p')$ as required.

Condition 2(b): We are required to prove that if $q^0_{p'} = after_{p'}(\sigma_1)$ is a non-leaf and all states of p' reached by transitions from $q^0_{p'}$ are leaves then we have $I \cap enabled(q^0_{p'}) = \emptyset$. Therefore, it is sufficient to prove that $a \in enabled(q^0_{p'})$ implies $a \in O \cup \{\ominus\}$. We will consider two cases.

First, let us suppose that $\sigma_1.a$ is a prefix of σ. Since p' is the union of p and σ and $\sigma_1.a$ reaches a leaf of p', we must have that $\sigma = \sigma_1.a$. By the definition of an STC, a is not an input as required.

Now consider the (second) case where $\sigma_1.a$ is not a prefix of σ. Since p' is the union of p and σ, we must have that $\sigma_1.a$ is a trace of p and so the state $q^0_p = after_p(\sigma_1)$ is a non-leaf. Since p' is the union of p and σ and all states of p' reached by transitions from $q^0_{p'}$ are leaves, we must have that all states of p reached by transitions from q^0_p are also leaves. Since p is non-redundant, we therefore have that a is not an input, as required. The result therefore follows.

Condition 2(c): Let us suppose that $\sigma_1 \in Ev(p')$ is such that $q^0_{p'} = after_{p'}(\sigma_1)$ is a non-leaf and all states of p' reached by transitions from $q^0_{p'}$ are leaves. We are required to prove that if $\ominus \in enabled(q^0_{p'})$ then $V(after_{p'}(\sigma_1.\ominus)) = fail$. We therefore assume that $\ominus$ is enabled at $q^0_{p'}$ and we consider two cases.

First, let us suppose that $\sigma_1.\ominus$ is a prefix of σ. Since p' is the union of p and σ and $\sigma_1.\ominus$ reaches a leaf of p', we must have that $\sigma = \sigma_1.\ominus$. By the definition of an STC, the state of p' reached by $\sigma_1.\ominus$ must have verdict *fail* as required.

Now consider the (second) case where $\sigma_1.\ominus$ is not a prefix of σ. Since p' is the union of p and σ, we must have that $\sigma_1.\ominus$ is a trace of p and so $q^0_p = after_p(\sigma_1)$ is a non-leaf. Since p' is the union of p and σ and all states of p' reached by transitions from $q^0_{p'}$ are leaves, we must have that all states of p reached by transitions from q^0_p are leaves. Since p is non-redundant, we must have that the

transition from q_p^0 with label $\ominus$ reaches a state with verdict *fail*. Thus, since p' is the union of p and σ we have that the transition from $q_{p'}^0$ with label $\ominus$ reaches a state with verdict *fail*. The result therefore follows. □

4.2. Conditions under which we can remove σ

We start by giving a condition under which we can simply remove σ and we do not have to change p (i.e. p is the union of p and σ).

Proposition 7. *Let us suppose that σ is an STC, p is a sound and non-redundant ATC and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. If $V(after_p(\sigma')) = fail$, then p is the union of p and σ.*

Proof. First note that since p is sound, by Definition 7 we have that σ' is a disallowed behaviour because it reaches a *fail* state. Further, by the definition of an STC, σ is a disallowed behaviour and all proper prefixes of σ are allowed behaviours. Since σ' is a prefix of σ and σ' is a disallowed behaviour, we must have that $\sigma' = \sigma$. We therefore know that σ is a failing evolution of p and so $Ev_F(p) = Ev_F(p) \cup Ev_F(\sigma)$.

It remains to prove that $Ev(p) = Ev(p) \cup Ev(\sigma)$. However, this follows from Proposition 3 which tells use that $Ev(C(\sigma)) \subseteq Ev(p)$. The result thus follows. □

Example 4. Consider the ATC p presented in Fig. 1 (right) and the STC $\sigma = !o_1 !o_1 !o_1 \ominus$. We have that p is the union of p and σ.

Note that the previous property is equivalent to the case where σ reaches a *fail* state when starting at the initial state of p.

4.3. Conditions under which we can combine test cases

We now consider the case where we cannot simply remove σ, leaving p unchanged, and consider when we can merge p and σ to form some p'. We will use σ' to denote the longest prefix of σ that is an evolution of p and q_p^0 to denote $after_p(\sigma')$. We will focus on the case where q_p^0 is not a leaf of p (in the next section we will see that if q_p^0 is a leaf of p then we can merge p and σ). We will consider three scenarios regarding the event in σ that follows σ':

- The next event in σ is an output (Proposition 8).
- The next event in σ is $\ominus$ (Proposition 9).
- The next event in σ is an input (Proposition 10).

We start by consider the case where the next event in σ is an output, with the following showing that this case cannot occur and so the proposed algorithm does not have to include rules for this case.

Proposition 8. *Let us suppose that σ is an STC, p is a sound and non-redundant ATC, and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. If the state $after_p(\sigma')$ is not a leaf and $\sigma' \neq \sigma$ then $next_\sigma(\sigma') \notin O$.*

Proof. We use proof by contradiction: we assume that $q_p^0 = after_p(\sigma')$ is not a leaf and $next_\sigma(\sigma') \in O$. By the definition of ATCs, $O \subseteq enabled(q_p^0)$. This contradicts σ' being the longest common prefix as required. □

The next two results give *necessary* conditions for there to be an ATC p' that is the union of p and σ. The next section gives rules for merging an ATC and an STC and provides a result (Proposition 13) that shows that the conditions below are also *sufficient* conditions.

We now consider the case where the next event in σ is $\ominus$.

Proposition 9. *Let us suppose that σ is an STC, p is a sound and non-redundant ATC and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. Let us suppose that $after_p(\sigma')$ is not a leaf and $next_\sigma(\sigma') = \ominus$. If there is an ATC p' that is the union of p and σ then $enabled(after_p(\sigma')) = O$.*

Proof. Let $q_p^0 = after_p(\sigma')$. First, since σ' is a maximal prefix of σ that is a trace of p, we have that $\ominus \notin enabled(q_p^0)$. By the definition of an ATC, since q_p^0 is not a leaf we have that $O \subseteq enabled(q_p^0)$. It is therefore sufficient to prove that no inputs are enabled at q_p^0.

We use proof by contradiction and assume that p' is the union of p and σ and some input $?i$ belongs to $enabled(q_p^0)$. Since $\sigma'.\ominus$ is a prefix of the trace σ, it is an evolution of the STC σ. Since p' is the union of p and σ, we therefore have that $\sigma'.\ominus$ is a trace of p'. In addition, every event enabled at the state q_p^0 of p is also enabled at the state $q_{p'}^0 = after_{p'}(\sigma')$. Thus, since $?i \in enabled(q_p^0)$, we also have that $?i \in enabled(q_{p'}^0)$. We therefore have that both $?i$ and $\ominus$ are enabled at state $q_{p'}^0$. This contradicts the definition of an ATC and so completes the proof by contradiction. □

Finally we have the case where the next event in σ is an input.

Algorithm 1 Checking whether an STC and an ATC can be merged.

```
function PossibleMerging
Input: σ, σ' ∈ Σ*_⊖, p = (Q, q_0, I, O, T, V), q_p^0 ∈ Q
if σ = σ' ∨ q_p^0 is a leaf then
    merge = true
else
    if (I ∪ {⊖}) ∩ enabled(q_p^0) = ∅ then
        merge = true
    else
        merge = false
    end if
end if
return merge
```

Proposition 10. *Let us suppose that σ is an STC, p is a sound and non-redundant ATC, and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. Let us suppose that $after_p(\sigma')$ is not a leaf of p and $next_\sigma(\sigma') = ?i$. If there is an ATC p' that is the union of p and σ then $enabled(after_p(\sigma')) = O$.*

Proof. Let q_p^0 be $after_p(\sigma')$. Let us suppose that there is some ATC p' that is the union of p and σ. Let $q_{p'}^0$ be $after_{p'}(\sigma')$. Consider the set of events $enabled(q_{p'}^0)$.

- $?i \in enabled(q_{p'}^0)$ since $next_\sigma(\sigma') = ?i$.
- All outputs must be enabled.
- If an event $a \in I \cup \{\ominus\}$ belongs to $enabled(q_p^0)$ then a must also belong to $enabled(q_{p'}^0)$. This is because all evolutions of p are also evolutions of p'.

The result now follows by observing that, since p' is an ATC, if input $?i \in enabled(q_{p'}^0)$, then we must have that $\ominus \notin enabled(q_{p'}^0)$ and also no other input is enabled at state $q_{p'}^0$. □

5. Merging STCs

This section develops the algorithm for converting a set S of STCs into an equivalent set A of ATCs. We start with an algorithm that checks whether an ATC p and an STC can be merged (Section 5.1). Section 5.2 then gives the rules for merging and proves that they are correct. Finally, the algorithm is given in Section 5.3.

5.1. Checking whether merging can happen

Algorithm 1 checks whether an STC σ can be merged with a sound and non-redundant ATC p. In this algorithm, as usual, σ' is the longest prefix of σ that is an evolution of p and $q_p^0 = after_p(\sigma')$. As noted above (Proposition 7), we only consider the case where $V(q_p^0) = ok$; if $V(q_p^0) = fail$ then we can simply remove σ without changing p. The algorithm allows merging to happen if one of four situations occurs:

1. $\sigma' = \sigma$ and $V(q_p^0) = ok$. In Section 5.2, Proposition 11 shows how we can form a suitable p'.
2. State q_p^0 of p is a leaf. In Section 5.2, Proposition 12 shows how we can form a suitable p'.
3. By Proposition 9, we know that if q_p^0 is not a leaf, $\sigma' \neq \sigma$, and $next_\sigma(\sigma') = \ominus$ then we can only merge p and σ' if $enabled(q_p^0) = O$.
4. By Proposition 10, we know that if q_p^0 is not a leaf, $\sigma' \neq \sigma$, and $next_\sigma(\sigma') \in I$ then we can only merge p and σ' if $enabled(q_p^0) = O$.

5.2. Rules for merging

We now consider the separate cases, giving results showing how an STC and an ATC can be combined. We use σ, σ' and q_p^0 as before. We consider three cases.

- We have that $\sigma' = \sigma$ and $V(q_p^0) = ok$ (Proposition 11).
- We have that $\sigma' \neq \sigma$, q_p^0 is a leaf and $V(q_p^0) = ok$ (Proposition 12).
- We have that $\sigma' \neq \sigma$, q_p^0 is not a leaf and $V(q_p^0) = ok$ (Proposition 13).

We start with the case where $\sigma' = \sigma$.

Proposition 11. *Let us suppose that σ is an STC and p is a sound and non-redundant ATC. Let us suppose that $\sigma \in Ev(p)$ and that $V(after_p(\sigma)) = ok$. Then, $after_p(\sigma)$ is a leaf and if we form a new ATC p' by changing the verdict of $after_p(\sigma)$ to $fail$ then p' is the union of p and σ.*

Proof. First, since σ is an STC, by Definition 1, σ must be a disallowed trace. Since σ is a disallowed trace and p is non-redundant, we must have that $after_p(\sigma)$ is a leaf of p. This has established the first part of the result. In addition, it immediately follows that $Ev(p') = Ev(p)$.

We are now required to prove that $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$ and we do so by proving that if σ_1 is a trace then $\sigma_1 \in Ev_F(p')$ if and only if $\sigma_1 \in Ev_F(p) \cup Ev_F(\sigma)$. By the definition of $Ev_F(p')$, $\sigma_1 \in Ev_F(p')$ if and only if $V(after_{p'}(\sigma_1)) = fail$. By the definition of p', this is the case if either $V(after_p(\sigma)) = fail$ or $\sigma_1 = \sigma$. Thus, $\sigma_1 \in Ev_F(p')$ if and only if $\sigma_1 \in Ev_F(p) \cup \{\sigma\}$. The result now follows by observing that $Ev_F(\sigma) = \{\sigma\}$. □

Example 5. Consider the ATC p presented in Fig. 1 (right) and the STC $\sigma = !o_1 !o_2$. Let p' be equal to p but changing the verdict of the state reached after the sequence $!o_1 !o_2$ from *ok* to *fail*. Then, p' is the union of p and σ.

We now consider the case where $\sigma' \neq \sigma$ and q_p^0 is a leaf of p. We start by showing how we can extend p with σ in this case. Note that since q_p^0 is a leaf of p, p does not have any transitions from q_p^0.

Definition 16. Let us suppose that $p = (Q, q_0, I, O, T, V)$ is a sound and non-redundant ATC, $\sigma = a_1 \dots a_k \in \Sigma_{\ominus}^+$ is an STC, $\sigma' = a_1 \dots a_\ell$ is the longest prefix of σ such that $\sigma' \in Ev(p)$, $l < k$, and $q_p^0 = after_p(\sigma')$. Further, suppose that q_p^0 is a leaf of p. The extension of p with σ, $\mathcal{E}xt_L(p, \sigma)$, is the ATC $p' = (Q', q_0, I, O, T', V')$ in which we have the following.

1. q_ℓ denotes q_p^0.
2. $Q' = Q \cup \{q_{\ell+1}, \dots, q_k\} \cup \{q_{i+1}^j \mid \ell \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
3. $T' = T \cup \{(q_i, a_{i+1}, q_{i+1}) \mid \ell \leq i < k\} \cup \{(q_i, !o_j, q_{i+1}^j) \mid \ell \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
4. The verdict function is defined as follows

$$V'(q) = \begin{cases} V(q) & \text{if } q \in Q \\ fail & \text{if } q = q_k \\ ok & \text{otherwise} \end{cases}$$

Example 6. Consider the ATC p in Fig. 1 (left) and the STC $\sigma = ?i_1 !o_1 \ominus$. Fig. 2 (left) gives the ATC $\mathcal{E}xt_L(p, \sigma)$.

Proposition 12. *Let us suppose that σ is an STC, p is a sound and non-redundant ATC and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. Let q_p^0 be $after_p(\sigma')$. Let us suppose that $V(q_p^0) = ok$, $\sigma \neq \sigma'$, and q_p^0 is a leaf of p. Then $p' = \mathcal{E}xt_L(p, \sigma)$ is the union of p and σ.*

Proof. First, by construction it is clear that p' satisfies the requirements of an ATC and $Ev(p') = Ev(p) \cup Ev(\sigma)$.

We now prove that $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$. Let σ_1 be a trace and it is sufficient to prove that $\sigma_1 \in Ev_F(p')$ if and only if $\sigma_1 \in Ev_F(p) \cup Ev_F(\sigma)$. Observe that $\sigma_1 \in Ev_F(p')$ if and only if $V(after_{p'}(\sigma_1)) = fail$. By the definition of p', this is the case if and only if either $V(after_p(\sigma_1)) = fail$ or $\sigma_1 = \sigma$. The first case holds if and only if we have that $\sigma_1 \in Ev_F(p)$. The second case holds if and only if we have that $\sigma_1 \in Ev_F(\sigma)$. Thus, $\sigma_1 \in Ev_F(p)$ if and only if either $\sigma_1 \in Ev_F(p) \cup Ev_F(\sigma)$ as required. □

Finally, we now consider the case where $\sigma' \neq \sigma$ and q_p^0 is *not* a leaf of p. We again extend p with σ but we require a slightly different extension function because p has transitions leaving q_p^0. The only difference between this definition and Definition 16 is that, because q_p^0 is not a leaf, we no longer need to add transitions from q_p^0 for alternative outputs. This is reflected in the second sets, used in defining Q' and T', having the condition $\ell + 1 \leq i < k$ rather than $\ell \leq i < k$.

Definition 17. Let us suppose that $p = (Q, q_0, I, O, T, V)$ is an ATC, $\sigma = a_1 \dots a_k \in \Sigma_{\ominus}^+$ is an STC, $\sigma' = a_1 \dots a_\ell$ is the longest prefix of σ such that $\sigma' \in Ev(p)$, $l < k$, and q_p^0 is $after_p(\sigma')$. Further, suppose that q_p^0 is not a leaf of p. The extension of p with σ, $\mathcal{E}xt_N(p, \sigma)$, is the ATC $p' = (Q', q_0, I, O, T', V')$ in which we have the following.

1. q_ℓ denotes q_p^0.
2. $Q' = Q \cup \{q_{\ell+1}, \dots, q_k\} \cup \{q_{i+1}^j \mid \ell + 1 \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
3. $T' = T \cup \{(q_i, a_{i+1}, q_{i+1}) \mid \ell \leq i < k\} \cup \{(q_i, !o_j, q_{i+1}^j) \mid \ell + 1 \leq i < k \wedge !o_j \in O \setminus \{a_{i+1}\}\}$.
4. The verdict function is defined as follows

$$V'(q) = \begin{cases} V(q) & \text{if } q \in Q \\ fail & \text{if } q = q_k \\ ok & \text{otherwise} \end{cases}$$

Example 7. Consider the ATC p presented in Fig. 1 (left) and the STC $\sigma = ?i_1 \ominus ?i_1 !o_1$. Fig. 3 presents the ATC $\mathcal{E}xt_N(p, \sigma)$.

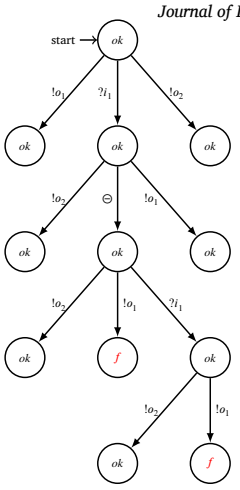


Fig. 3. The ATC $\mathcal{E}xt_N(p, ?i_1 \ominus ?i_1 !o_1)$ for the ATC p in Fig. 1 (left).

The proof of the following is almost identical to that of Proposition 12.

Proposition 13. *Let us suppose that σ is an STCs, p is an ATC and σ' is the longest prefix of σ such that $\sigma' \in Ev(p)$. Let q_p^0 be $after_p(\sigma')$. Let us suppose that $V(q_p^0) = ok$, $\sigma \neq \sigma'$, and q_p^0 is not a leaf of p. Further, suppose that $enabled(q_p^0) = O$. Then the ATC $p' = \mathcal{E}xt_N(p, \sigma)$ is the union of p and σ.*

5.3. Algorithm

We now give an algorithm for combining a set of STCs. This algorithm maintains a set of ATCs, which is initially empty, and then goes through a number of iterations, adding one STC in each iteration. When an STC σ is considered, the algorithm determines whether σ can be merged with any of the current ATCs: if σ can be merged with one or more of the ATCs then it is merged with one of these and otherwise the ATC $C(\sigma)$ is added to the current set of ATCs.

Let p be an ATC, σ be an STC and σ' be the longest prefix of σ such that $\sigma' \in Ev(p)$. Further, let q_p^0 be $after_p(\sigma')$. Note that, since p is deterministic, the sequence reaches a single state. Several cases are considered in the algorithm and these are now briefly reviewed.

1. If $V(q_p^0) = fail$, then Proposition 7 tells us that we can discard σ. Note that, as a result of the definition of STCs, this scenario can only occur if $\sigma = \sigma'$.
2. If $V(q_p^0) = ok$ and $\sigma = \sigma'$, then Proposition 11 tells us that q_p^0 is a leaf of p and we simply change the verdict of q_p^0 to $fail$. The idea is that if we observe σ then we must indicate that this behaviour is not allowed and so the verdict must be $fail$.
3. If $V(q_p^0) = ok$ and $\sigma \neq \sigma'$ then there are the following cases:
   a. q_p^0 is a leaf of p. By Proposition 12, we can just extend p with the rest of σ. Thus, one can merge σ and p to form $p' = \mathcal{E}xt_L(p, \sigma)$.
   b. q_p^0 is not a leaf of p and $next_\sigma(\sigma') \in O$. By Proposition 8 we know that this case cannot occur since, by the definition of ATCs, $O \subseteq enabled(q_p^0)$, contradicting σ' being the longest common prefix.
   c. q_p^0 is not a leaf of p and $next_\sigma(\sigma') = \ominus$. Proposition 9 tells us that we can only merge p and σ if $enabled(q_p^0) = O$. If we can merge p and σ then Proposition 13 tells us that we can extend p with the rest of σ to form $p' = \mathcal{E}xt_N(p, \sigma)$.
   d. q_p^0 is not a leaf of p and $next_\sigma(\sigma') \in I$. By Proposition 10, we can only merge p and σ if $enabled(q_p^0) = O$. If we can merge p and σ then Proposition 13 tells us that we just extend p with the rest of σ to form $p' = \mathcal{E}xt_N(p, \sigma)$.

We now give the overall algorithm (Algorithm 2) that takes a set of sequential test cases and produces a set of ATCs. The algorithm is iterative. In each iteration, the algorithm tries to merge an STC σ with an ATC that includes a non-empty prefix of σ. The algorithm uses the $PossibleMerging$ function (Algorithm 1) to determine which of the current ATCs can be merged with σ, placing these in the set $ATCsig$. If $ATCsig$ is non-empty then the algorithm chooses an ATC p from $ATCsig$ and replaces p with the ATC p' formed by merging p and σ. If $ATCsig$ is empty, and so the STC cannot be merged with any ATC, then a new ATC corresponding to $C(\sigma)$ will be created. Naturally, this last case applies on the first iteration since at this point there are no ATCs.

In an iteration, it is possible that $ATCsig$ contains more than one ATC that can be merged with σ. The algorithm chooses an ATC p from $ATCsig$ such that this maximises the length of the longest prefix of σ that is in the set of evolutions of p (i.e. the length of the corresponding σ'). The motivation is that this minimises the increase in the total number of nodes in the ATCs (and maximises how much the traces of p and σ overlap).

The algorithm uses a separate function $Merge$ (see Algorithm 3) that takes an ATC p and a sequential test σ and combines these.

Algorithm 2 Creating ATCs from STCs.

Input: $\Sigma = I \cup O$, $STCs \subseteq \Sigma_{\ominus}^{+}$
Let $ATCs = \emptyset$
for all $\sigma = a_1 \dots a_n \in STCs$ **do**
  $ATCsig = \{p \mid p \in ATCs \wedge \exists \sigma' \in (pref(\sigma) \setminus \{\epsilon\}) \cap Ev(p)\}$
  $merge = false$
  while $(\neg merge \wedge ATCsig \neq \emptyset)$ **do**
    Choose a tree $p = (Q, q_0, I, O, T, V) \in ATCsig$ including
    the longest non-empty prefix of σ. Let $\sigma' = a_1 \dots a_k$ be this prefix.
    $q_p^0 = after_p(\sigma')$
    $ATCsig = ATCsig \setminus \{p\}$
    if $V(q_p^0) = fail$ **then**
      $merge = true$
    else if $PossibleMerging(\sigma, \sigma', p, q_p^0)$ **then**
      $p' = Merge(\sigma, \sigma', p, q_p^0)$
      $ATCs = ATCs \setminus \{p\} \cup \{p'\}$
      $merge = true$
    end if
  end while
  if $\neg merge$ **then**
    $ATCs = ATCs \cup \{C(\sigma)\}$
  end if
end for

Algorithm 3 Merging an STC and an ATC.

function Merge
Input: $\sigma = a_1 \dots a_n, \sigma' = a_1 \dots a_k \in \Sigma_{\ominus}^{*}$, $p = (Q, q_0, I, O, T, V), q_p^0 \in Q$
if $\sigma' = \sigma$ **then**
  Update V with $V(q_p^0) = fail$
else
  if q_p^0 is a leaf of p **then**
    $p' = \mathcal{E}xt_L(p, \sigma)$
  else
    $p' = \mathcal{E}xt_N(p, \sigma)$
  end if
end if
return p'

Example 8. Consider $I = \{?i_1, ?i_2\}$, $O = \{!o_1, !o_2\}$. Fig. 4 provides a graphical representation of how the algorithm produces a set of ATCs when it receives as input the set $S = \{!o_1?i_1!o_2\ominus, !o_1?i_1!o_1, ?i_2!o_1, !o_1?i_1\ominus, ?i_2\ominus, ?i_2!o_2!o_1\}$. The algorithm goes through the following iterations.

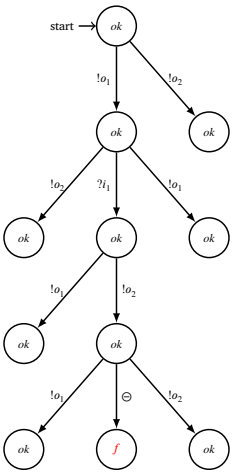
1. In the first iteration the set of ATCs is empty, therefore a new ATC p_1 corresponding to $C(!o_1?i_1!o_2\ominus)$ is created (Fig. 4a).
2. Next, considering the STC $\sigma = !o_1?i_1!o_1$, the algorithm searches for the longest prefix of σ that is an evolution of p_1. In this case, this prefix coincides with σ, and the verdict of the state of p_1 reached by σ is changed to *fail* (Fig. 4b).
3. The algorithm now considers the STC $?i_2!o_1$, none of its prefixes are evolutions of p_1 and the ATC $p_2 = C(?i_2!o_1)$ is created (Fig. 4c).
4. The longest prefix of $!o_1?i_1\ominus$ in p_1 and p_2 are $!o_1?i_1$ and $!o_1$, respectively. In the first case, the outgoing transitions from the state reached by the prefix are not labelled with an input or $\ominus$ and the state reached by the second in p_2 is a leaf. Therefore, the STC can be merged with either of them. Since $!o_1?i_1$ is longer than $!o_1$, p_1 is selected to be merged with the STC and is extended with the rest of the sequence, that is, $\ominus$ (Fig. 4d).
5. The STC $?i_2\ominus$ can be merged only with p_2 by extending p_2 with $\ominus$ to form $\mathcal{E}xt_N(p_2, ?i_2\ominus)$ (Fig. 4e).
6. Finally, the STC $?i_2!o_2!o_1$ is also merged with p_2. In this case, the state reached by the prefix $?i_2!o_2$ is a leaf and the merge of p_2 and the STC corresponds to $\mathcal{E}xt_L(p_2, ?i_2!o_2!o_1)$ (Fig. 4f).

In conclusion, the set S is represented by the ATCs given in Figs. 4d and 4f.

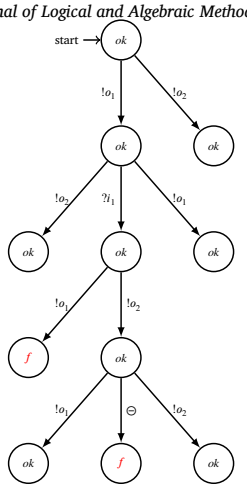
Finally, the correctness of Algorithm 2 is based on the merging rules being correct.

Theorem 1. *If Algorithm 2 is given set S of STCs as input then it returns a set A of sound and non-redundant ATCs such that A is equivalent to S.*

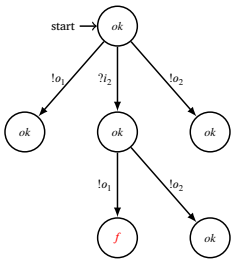
Proof. First, note that each iteration of the algorithm reduces the number of STCs in $STCs$ by 1 and $STCs$ starts with S. Since S is finite, we know that the algorithm terminates with $S = \emptyset$.



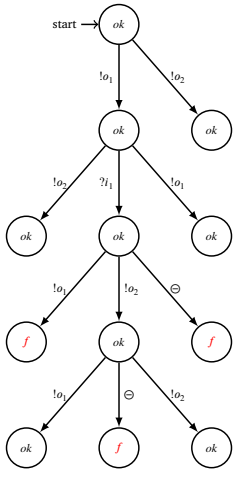
(a) ATC p_1.



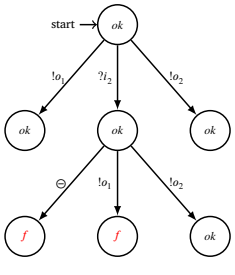
(b) p_1 after merging with $\sigma = !o_1?i_1!o_1$.



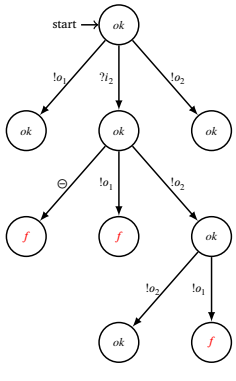
(c) ATC p_2.



(d) p_1 after merging with $!o_1?i_1\ominus$.



(e) p_2 after merging with $?i_2\ominus$.



(f) p_2 after merging with $?i_2!o_2!o_1$.

Fig. 4. ATCs obtained from the application of Algorithm 2.

Now consider one iteration of the algorithm: to prove correctness of the algorithm as a whole it is sufficient to prove that if an iteration replaces non-redundant ATC $p \in ATCs$ and STC σ with ATC p' then p' is the union of p and σ. We separately consider the cases where merging happens in Algorithm 2 and what has been established about these.

Case 1: $V(q_p^0) = ok$ and $\sigma = \sigma'$. Algorithm 2 forms p' by changing the verdict of q_p^0 to *fail*. Proposition 11 tells us that p' is the union of p and σ.

Case 2: $V(q_p^0) = ok$, $\sigma \neq \sigma'$, and q_p^0 is a leaf of p. Algorithm 2 merges σ and p to form $p' = \mathcal{E}xt_L(p, \sigma)$. By Proposition 12, p' is the union of p and σ.

Case 3: $V(q_p^0) = ok$, $\sigma \neq \sigma'$, q_p^0 is not a leaf of p, and $enabled(q_p^0) = O$. In this case, Algorithm 2 extends p with the rest of σ to form $p' = \mathcal{E}xt_N(p, \sigma)$. Proposition 10, tells us that p' is the union of p and σ.

Thus, whenever Algorithm 2 merges an ATC p and STC σ to form an ATC p' we have that p' is the union of p and σ. By the definition of union, we have that $Ev(p') = Ev(p) \cup Ev(\sigma)$ and $Ev_F(p') = Ev_F(p) \cup Ev_F(\sigma)$. Thus, each step of the algorithm preserves the set of evolutions and the set of failing evolutions and so A is equivalent to S. Finally, Propositions 5 and 6 tell us that the ATCs formed are sound and non-redundant. The result thus follows. □

This tells us that if S is capable of showing that a given SUT is faulty then A can also achieve this and, in addition, the ATCs in A are sound and non-redundant.

6. Conclusions and future work

In this paper we considered a testing framework where test requirements are given by a set of *negative* test cases: a sequence $\sigma.a$ represents the situation in which, if the SUT executes the sequence σ then a should not be observed next. The goal of the work described in this paper was to provide an algorithm that transforms such a set of test cases (that we called sequential test cases) into a set of adaptive test cases that encode the same information in a more compact way. The motivation for this work was that the use of ATCs improves the efficiency of the testing process because there are situations where testing with ATCs will continue while testing with an STC would require the testing process to be *reset*. We formally proved the correctness of the algorithm and presented a complete example to show some of its intricacies.

There are several possible lines of future work. In this work we considered two verdicts: *ok* and *fail*. In contrast, a test case could have states that have verdict *inconclusive*, denoting the test objective not having been achieved. We potentially lose some information if we replace *inconclusive* verdicts. An alternative approach would have been for a state of an ATC p to be associated with a tuple of verdicts, one for each STC used in forming p. Such a verdict would provide information about which test objectives have been achieved and also which original STCs failed when a *fail* verdict is given; such information could help with traceability. We would only require small changes to our approach to introduce such tuples of verdicts.

The proposed algorithm is iterative and, in an iteration, an STC under consideration is merged with at most one ATC. This makes sense if we are interested in limiting the (sum of) the sizes of the ATCs. Consider, however, the case where we have ATCs p_1, where all traces start with $!o_1.!o_2.?i_1$, and p_2, where all traces start with $!o_1.!o_2.?i_2$. Further, let us suppose that we are considering the STC $\sigma = !o_1.!o_2.!o_3.!o_4$. There is potential to merge σ with *both* of the ATCs, to form two new ATCs that we call p_1' and p_2'. If we do this then the test objective (STC) σ can be achieved when using either p_1' or p_2' and so we might require fewer test executions in order to achieve all test objectives. It would be straightforward to update the algorithm to allow such an approach.

A final observation is that if we have access to the specification then we might revisit the decision to give an *ok* verdict if the tester is waiting for output $!o_1$ after some prefix σ_1 of σ and the SUT instead produces output $!o_2$. For example, when completing an STC to form an ATC, we could check each such $\sigma_1.!o_2$ against the specification and set the verdict to be *fail* if $\sigma_1.!o$ is a disallowed trace.

CRediT authorship contribution statement

Robert M. Hierons: Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Funding acquisition, Conceptualization. **Mercedes G. Merayo:** Writing – review & editing, Methodology, Writing – original draft, Funding acquisition, Conceptualization, Formal analysis. **Manuel Núñez:** Writing – review & editing, Funding acquisition, Formal analysis, Writing – original draft, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: None.

Data availability

No data was used for the research described in the article.

References

[1] J. Baxter, A. Cavalcanti, M. Gazda, R.M. Hierons, Testing using CSP models: time, inputs, and outputs, ACM Trans. Comput. Log. 24 (2) (2023) 17:1–17:40.
[2] R. Lefticaru, R.M. Hierons, M. Núñez, Implementation relations and testing for cyclic systems with refusals and discrete time, J. Syst. Softw. 170 (110738) (2020) 1–20.

[3] M. Núñez, R.M. Hierons, R. Lefticaru, Implementation relations and testing for cyclic systems: adding probabilities, Robot. Auton. Syst. 165 (2023) 104426.
[4] A.R. Cavalli, T. Higashino, M. Núñez, A survey on formal active and passive testing with applications to the cloud, Ann. Télécommun. 70 (3–4) (2015) 85–93.
[5] M.-C. Gaudel, "Testing can be formal too": 30 years later, in: B. Meyer (Ed.), The French School of Programming, Springer, 2024, pp. 17–45.
[6] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, H. Zedan, Using formal specifications to support testing, ACM Comput. Surv. 41 (2) (2009) 9:1–9:76.
[7] A. Cavalcanti, M. Gaudel, Testing for refinement in CSP, in: 9th Int. Conf. on Formal Methods and Software Engineering, ICFEM'07, in: LNCS, vol. 4789, Springer, 2007, pp. 151–170.
[8] A. Cavalcanti, R.M. Hierons, S.C. Nogueira, Inputs and outputs in CSP: a model and a testing theory, ACM Trans. Comput. Log. 21 (3) (2020) 24:1–24:53.
[9] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y.L. Traon, M. Harman, Mutation Testing Advances: An Analysis and Survey, Advances in Computers, vol. 112, Elsevier, 2019, pp. 275–378.
[10] A. Cavalcanti, J. Baxter, R.M. Hierons, R. Lefticaru, Testing robots using CSP, in: 13th Int. Conf. on Tests and Proofs, TAP'19, in: LNCS, vol. 11823, Springer, 2019, pp. 21–38.
[11] C.M. Poskitt, Y. Chen, J. Sun, Y. Jiang, Finding causally different tests for an industrial control system, in: 45th IEEE/ACM Int. Conf. on Software Engineering, ICSE'23, IEEE, 2023, pp. 2578–2590.
[12] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, RoboChart: modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. 18 (5) (2019) 3097–3149.
[13] R.J. Somers, J.A. Douthwaite, D.J. Wagg, N. Walkinshaw, R.M. Hierons, Digital-twin-based testing for cyber-physical systems: a systematic literature review, Inf. Softw. Technol. 156 (2023) 107145.
[14] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines: a survey, Proc. IEEE 84 (8) (1996) 1090–1123.
[15] U.C. Türker, R.M. Hierons, G.D. Barlas, K. El-Fakih, Incomplete adaptive distinguishing sequences for non-deterministic FSMs, IEEE Trans. Softw. Eng. 49 (9) (2023) 4371–4389.
[16] J. Tretmans, Model based testing with labelled transition systems, in: Formal Methods and Testing, in: LNCS, vol. 4949, Springer, 2008, pp. 1–38.
[17] A. Fellner, M. Tabaei Befrouei, G. Weissenbacher, Mutation testing with hyperproperties, Softw. Syst. Model. 20 (2021) 405–427.
[18] D. Basile, M.H.t. Beek, S. Lazreg, M. Cordy, A. Legay, Static detection of equivalent mutants in real-time model-based mutation testing, Empir. Softw. Eng. 27 (2022) 160.
[19] C. Mandrioli, S. Yeob Shin, M. Maggio, D. Bianculli, L.C. Briand, Stress testing control loops in cyber-physical systems, ACM Trans. Softw. Eng. Methodol. 33 (2) (2024) 35.