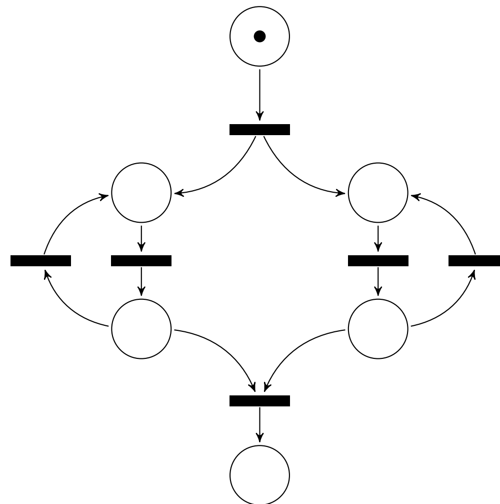
Automatically Tracking Model Based Testing of Asynchronously Communicated Protocol Implementations

An MSc Thesis



Author
Rick Klomp
ICA-5540232

Supervisors
dr. S.W.B. Prasetya
dr. J. Hage

Abstract

Automatic testing of protocol implementations given only a formal protocol specification has long since been achieved for synchronously communicated protocols. However, acquiring similar results under asynchronous communication is not straightforward and has not been practically achievable. This work solves state-space explosions due possible interleaving of events under asynchronous communication through inference optimization from a refined concept of soundness. Additionally, the high concurrent nature of asynchronously communicating systems necessitated a shift from formally specifying protocols using Labeled Transition Systems to formally specifying protocols using Petri Nets. An EDSL for describing protocols in parallel composed structures has been defined that translates to Petri Net structure. These achievements have resulted in a first concrete fundamental foundation to enable practical application of automatic testing theory to asynchronously communicated protocols.

Utrecht University

February 9, 2018

CONTENTS

1 Introduction 4
- 1.1 Modeling Protocol Specifications . . . 5
- 1.2 Testing Synchronous Communication Protocols . . . 6
- 1.3 Testing Asynchronous Communication Protocols . . . 7

2 The Consequences of Asynchronicity 9
- 2.1 Inference Inaccuracy Decreases Test Strategy Efficiency . . . 9
- 2.2 Async (μ_s) Increases Inaccuracy . . . 10
- 2.3 Parallel Segments Increase Inaccuracy . . . 11
- 2.4 Sound Conversation States cannot Produce Sound Error Reports . . . 12

3 A Sound and Accurate APSL Engine 12
- 3.1 Petri Nets . . . 12
- 3.2 Parallel Segments Revisited . . . 15
- 3.3 θ transitions . . . 15
- 3.4 Relationships and Invariants over Values . . . 17

4 Petri Nets EDSL 17
- 4.1 PNSL . . . 20

5 Case Study: WebSocket Echo Server 21
- 5.1 The Specification . . . 21
- 5.2 Test Results . . . 23

6 Related Work 24
- 6.1 Automatic Test Tools . . . 24
- 6.2 Formal Specification of Asynchronous Communication Protocols . . . 24

7 Future Work 25
- 7.1 Test Strategy . . . 25
  - 7.1.1 Testing for Operational Reliability . . . 25
  - 7.1.2 Testing for Failures . . . 26

8 Conclusion 27

References 27

Appendix A 29

1 INTRODUCTION

When multiple systems communicate with each other according a certain protocol (e.g. HTML for rendering web pages), it is important that both follow the rules that have been defined by that protocol. However, specifications of protocols are sometimes ambiguous or unclear. Even with clearly and unambiguously specified protocols it remains difficult to create a system that communicates conforming to the protocol. Flaws in an implementation can give rise to unexpected system behavior and might enable attack paths targetable for malicious intent.

It is thus desired to verify implementations due to an inherent proneness to errors and the necessity of preventing bugs. A well known technique that is applied to locate errors involves a test environment that runs a set of tests and compares the observed results with what was expected. However, it is difficult and time consuming to manually construct a set of test cases that cover a significant part of the implemented protocol's use cases. Furthermore, implementing the test environment and the test cases is itself an error prone task.

A lot of research has been performed in pursuit of decreasing the engineering effort, focusing on enabling automatic testing the implementation of a protocol using Model Based Testing (MBT) techniques.

Recent work of Tervoort[1] has achieved promising results. The framework that he has established allows for automatically testing implementations of protocols. It provides a dedicated language that can be used to describe the protocol (*Another Protocol Specification Language*, or APSL). There exist plenty of other tools that can automatically test protocol implementations. However, the novelty of APSL is its message description sublanguage (*Another Message Specification Language*, or AMSL), which allows for fine-grained (i.e. up to binary encodings) specification of each message type. This is an essential ingredient for an automatic test tool if we want to establish true confidence in an implementation. With AMSL, APSL is not limited to only testing the transitions from states to states, but can also purposefully test transitions with specific values assigned to the variables within messages. In addition, AMSL enables tracking of what values have been assigned to each variable inside a message during a test.

One such purpose is the mimicking of operational behavior. By assigning values to variables that reflect real system behavior (and randomly distributed over a distribution that follows operational behavior), the tester can automatically test the implementation under realistic circumstances. Another such purpose is the specific testing of error prone message instances (this could for example include assigning values around minimum or maximum allowed borders). We argue that establishing a certain coverage over the possible instances of each message type is an essential goal that needs to be achieved in order to obtain some confidence in the implementation. If we are satisfied with a coverage that only depicts a percentage of transitions in global state that have been performed during testing of the protocol, then we neglect that bugs often only arise under certain combinations of values within a message instance. This aspect of MBT has mostly been ignored in the literature and offers a lot of potential beyond what we have done. By embedding the specification of message types into the APSL tool we are able to fully control which instances of messages are generated, which to the best of our knowledge has not been done by any existing work.

We did not delve further into how to best utilize this recently enabled aspect of MBT, this is left for future work. APSL was only able to test synchronously communicated protocol implementations. In fact, none of the currently available MBT tools can effectively test any asynchronously communicated protocol implementation based solely on a model of the protocol. The contributions of this work follow from a pursuit to enabling MBT to handle asynchronously communicated protocols.

Asynchronous communication suits itself to communication between concurrent processes (i.e. more so compared to synchronous communication). The concurrent nature in the sending and receiving of messages allows asynchronous protocols complete freedom in specifying when synchronization is necessary and when it can be neglected. Consequently, asynchronous protocols often utilize this by specifying communication stages wherein multiple segments are traversed without (or with little) synchronization between each segment. By doing this, the protocol enables reaching higher bandwidth throughput, at the expense of decreased control. Additionally, it can be utilized to specify unrelated mechanisms without them affecting each others traversal. For example, a protocol might specify two concurrently traversed segments A and B, where A specifies the communication of control messages and B specifies a keep alive mechanism. No matter how far the communication has traversed into the control segment, according to the specification the communication can always continue the traversal of the keep alive segment, and vice versa.

We were not only challenged by the concurrent nature in the sending and receiving of messages but also by patterns of concurrent segments in protocols. Not only does it feel unnatural to specify these concurrent patterns of protocols using the Labeled Transition System formalism (which is the conventional approach in MBT), it apparently also makes it much more difficult (and perhaps impossible) to effectively apply testing theory to these protocols when specified with an LTS.

This work contributes the following to the research area.

- We define a new perspective to the automatic tracking of communication. From this perspective a novel optimization in state inference is found (Section 2). The optimization, in confluence with the following contribution, is shown to be an essential ingredient for *feasible* automatic testing of asynchronously communicated protocol implementations.
- To reach effective leverage of the optimization, concurrent segments need to be represented explicitly in the specification. The conventional LTS can only represent these implicitly by interleaving segments' states, losing explicit information about the concurrent behavior. We propose an MBT approach where protocols are specified using the Petri Net formalism (Section 3). To a large extent this formalism builds on top of the well known LTS based formalism for MBT.

- We define an EDSL for specifying protocols that translate to a Petri Net data structure (Section 4). This EDSL leverages common patterns that are present in asynchronous protocols to make the process of specifying a protocol feel natural and less error prone.

The following subsections introduce a set of concepts and formalisms (among which the LTS based formalism for MBT) that have been acquired by related work and are essential stepping stones for the novel ideas and formalisms that are introduced in later sections. Section 2 describes an optimization technique concerning the tracking of asynchronous communication and shows that we cannot optimally apply it when using the conventional LTS based formalism to describe protocol specifications. Section 3 introduces a Petri Net based formalism for MBT which does succeed to optimally apply the optimization technique. Section 4 introduces an EDSL for describing protocol specifications in a structure compatible with Petri Nets. Section 5 presents a case study on an application of the modified APSL tool to the WebSocket protocol. Section 6 relates our work to prior research. Section 7 describes some interesting directions that can be further explored in future work. Finally, Section 8 concludes with what this work has achieved and established.

1.1 Modeling Protocol Specifications

APSL can automatically generate, execute and verify results of test cases that are applied to an implementation. It, as do related tools capable of doing at least one of these automatically, requires some formal specification method to derive essential information from the System Under Test (SUT). There is a general consensus in related work to formally specify protocols using Labeled Transition Systems (LTS) or some extension of this formalism. In [2], [3] test suites for synchronous communication protocols are automatically generated from the LTS. And [4] automatically derives test cases from the LTS, executes them and validates the results according to the LTS.

Labeled Transition Systems are an extension of Finite State Machines. A Finite State Machine (FSM) is an abstract machine (S, E) that is always in exactly one of its states. It consists of a finite set of states ($s \in S$) and a finite set of directed edges ($(s_i, s_j) \in E$). Looping edges are allowed (i.e. edges with $s_i == s_j$).

When modeling communication protocols, the edges of the FSM are labeled with a communication action (the sending of a message of some specified type and from whom to whom this message is sent), or the absence of a communication action (τ). An FSM that is extended with labels over its edges ($(s_i, a, s_j) \in E$) is a Labeled Transition System (LTS). Figure 1 presents an LTS that models a communication protocol. Occasionally a formula refers to the label of an edge e with e.ev.

Each non-τ labeled edge is traversed upon either an observable or controlled action, triggered by a message received from or sent to the SUT, respectively. The event of sending control message of type a is notated as $control_a$ or as $a_?$; the latter is typically used in formulas for brevity. And the event of receiving observe message of type x is notated as $observe_x$ or as $x_!$.

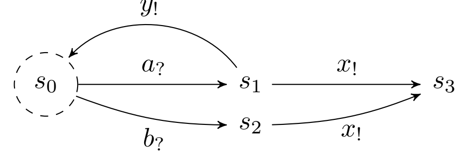


Fig. 1. A Labeled Transition System (LTS) consisting of four states. Each edge represents an allowed transition given either a certain observed action (at exclamation marked labels) or a controlled action (at question marked labels). The with stripes encircled state is the initial state of the LTS.

A (communication) trace is a communicated sequence of events. In this document, traces (and list structures in general) are shown with a dot between each event. An example trace is shown below.

$$a_? \cdot x_! \cdot z_!$$

A trace captures the events earliest to latest from left to right, respectively. That is, in the trace shown above $control_a$ was the first action that was registered by the tester, followed by $observe_x$, and $observe_z$ was the last action that was registered.

Consider for example the LTS presented in Fig. 1 and assume the communication trace $a_? \cdot y_! \cdot b_? \cdot x_!$. s_0 is the state that we started in (this is indicated by the striped circle around s_0 in the figure), and the communication reached s_3 after receiving $observe_x$ from the SUT.

The formulas below define some operators that are applicable to lists (these are applied in formulas to modify traces as well as other list structured data). ϵ denotes an empty list. $\preceq$ defines a precedence relation between the arguments xs and ys. This relation enforces that ys must either be equal to xs or be an extension of xs. $\underline{\setminus}$ mutates the first argument (xs) iteratively on each element a of the second argument, each time removing the first occurence of a in xs. $\underline{\cup}$ appends two lists together.

$$xs \preceq ys \overset{\textbf{def}}{=} \exists ys'.\ ys = xs \cdot ys'$$

$$\epsilon \underline{\setminus} a \overset{\textbf{def}}{=} \epsilon$$
$$a \cdot xs' \underline{\setminus} a \overset{\textbf{def}}{=} xs'$$
$$x \cdot xs' \underline{\setminus} a \overset{\textbf{def}}{=} x \cdot (xs' \underline{\setminus} a)$$

$$xs \underline{\setminus} \epsilon \overset{\textbf{def}}{=} xs$$
$$xs \underline{\setminus} y \cdot ys' \overset{\textbf{def}}{=} (xs \underline{\setminus} y) \underline{\setminus} ys'$$

$$xs \underline{\cup} \epsilon \overset{\textbf{def}}{=} xs$$
$$xs \underline{\cup} y \cdot ys' \overset{\textbf{def}}{=} y \cdot xs \underline{\cup} ys'$$

Additionally, the following symbols concerning traces and characteristics of a specification are used in the sequel.

$$L \overset{\textbf{def}}{=} \{a \mid \exists e.\ e \in E \ \wedge\ e.\mathsf{ev} = a\}$$
$$L_{\mathsf{c}} \overset{\textbf{def}}{=} \{a_? \mid \forall a_?.\ a_? \in L\}$$

$$L_{\mathsf{o}} \stackrel{\text{def}}{=} \{x_! \mid \forall x_!.\ x_! \in L\}$$
$$L^* \stackrel{\text{def}}{=} \{a_0 \cdot .. \cdot a_n \mid \{a_0, .., a_n\} \subseteq L\}$$
$$L_{\mathsf{c}}^* \stackrel{\text{def}}{=} \{a_0 \cdot .. \cdot a_n \mid \{a_0, .., a_n\} \subseteq L_{\mathsf{c}}\}$$
$$L_{\mathsf{o}}^* \stackrel{\text{def}}{=} \{a_0 \cdot .. \cdot a_n \mid \{a_0, .., a_n\} \subseteq L_{\mathsf{o}}\}$$
$$\sigma \stackrel{\text{def}}{=} \exists evs \in L^*.\ evs$$
$$\sigma_{\mathsf{ctrl}} = \sigma_{\mathsf{c}} \stackrel{\text{def}}{=} \exists evs \in L_{\mathsf{c}}^*.\ evs$$
$$\sigma_{\mathsf{obs}} = \sigma_{\mathsf{o}} \stackrel{\text{def}}{=} \exists evs \in L_{\mathsf{o}}^*.\ evs$$
$$a \cdot * \stackrel{\text{def}}{=} a \cdot L^*$$

Some formulae are defined using an **implies** keyword. This keyword (a **implies** b) captures the following relation: if a holds, then from this follows that b also holds. Similarly, (a **we have** b) is employed that captures the following relation: for all instances of a, b must hold.

APSL updates, after each registered event, where the two participants (the SUT and the tester itself) are in the protocol according to the LTS specification. APSL was only capable of performing this inference process soundly when communicating synchronously with the SUT.

The following subsection introduces the process of tracking the state under synchronous communication. The subsequent subsection introduces the additional aspects that need to be considered when upgrading this process to enable tracking the state under asynchronous communication.

1.2 Testing Synchronous Communication Protocols

The act of communicating a message synchronously between a set of participants occurs simultaneously by all participants. When one participant is in the act of sending a message, all other participants are in the act of receiving that message. To enable this, the protocol is required to thoroughly specify who has the right to send at each step of the communication. There should be no ambiguity about when a participant must listen and when it is expected to send. For example, during a phone call two participants communicate synchronously (assuming a civil conversation at least). In order for meaningful communication to take place, a participant must actively listen while the other talks and vice versa. Many peripherals like mice and keyboards communicate synchronously with the computer.

In [5], Tretmans formalizes an equivalence relation that relates observed traces to LTS specifications of synchronously communicating protocols. When we are in state s_i and register an occurrence of event a, according the LTS we (possibly) traversed to state s_j when there is a direct path from s_i to s_j that consumes a exactly once.

$$s_i \xrightarrow[\text{Spec}]{a} s_j \stackrel{\text{def}}{=} (s_i, a, s_j) \in E$$

Notice that a here can actually denote any label, including τ which is traversed without consuming a registered event and is thus always traversable.

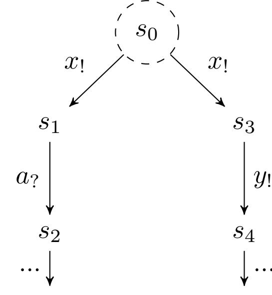


Fig. 2. An invalid specification $Spec_2$. This specification does not describe a sensible synchronous communication protocol due to the ambiguity concerning whom will send and whom will receive the next message after the tester receives $observe_x$.

An observed trace is related to the specification by consecutive application of this single event traversal relation.

$$s_i(\xrightarrow[\text{Spec}]{a})^+ s_j \stackrel{\text{def}}{=} s_i \xrightarrow[\text{Spec}]{a} s_j \ \vee$$
$$\exists s_i'.\ s_i(\xrightarrow[\text{Spec}]{a})^+ s_i' \xrightarrow[\text{Spec}]{a} s_j$$

$$s_i \xRightarrow[\text{Spec}]{\epsilon} s_j \stackrel{\text{def}}{=} s_j = s_i \ \vee\ s_i(\xrightarrow[\text{Spec}]{\tau})^+ s_j$$
$$s_i \xRightarrow[\text{Spec}]{a \in L} s_j \stackrel{\text{def}}{=} \exists s_i'\ s_i''.\ s_i \xRightarrow[\text{Spec}]{\epsilon} s_i' \xrightarrow[\text{Spec}]{a} s_i'' \xRightarrow[\text{Spec}]{\epsilon} s_j$$
$$s_i \xRightarrow[\text{Spec}]{a_0 \cdot a_1 \cdot .. \cdot a_n \in L^*} s_j \stackrel{\text{def}}{=} \exists s_i'.\ s_i \xRightarrow[\text{Spec}]{a_0} s_i' \xRightarrow[\text{Spec}]{a_1 \cdot .. \cdot a_n} s_j$$

Automatic validation of test case results uses this reachability operator to define which observations are according to specification and which are not. The implementation passes a test case when a communication trace σ is observed and $\exists s_j.\ s_i \xRightarrow[\text{Spec}]{\sigma} s_j$, where s_i is the initial state of the specification. If not, this test case has uncovered a mismatch between the implementation and the specification.

When testing a protocol that specifies synchronous communication, a tester can always determine a priori to the next occurring message if it is receiving or sending. However, it remains challenging to test these protocols automatically due to the non-deterministic behavior of the SUT. The SUT in Fig. 1 for example can either send $observe_x$ or $observe_y$ from s_1. Depending on which message it decides to send, the system either traverses to s_0 or to s_3. It is not apparent from the specification what this decision is based on. This non-deterministic behavior decreases the control that a tester has, and thereby increases the difficulty of executing some test strategy as intended (where a test strategy defines some goal to be reached during the test).

In a more extreme case a section of the LTS might reflect the protocol imprecisely such that at later steps it becomes ambiguous who should send the next message. Consider for example the LTS in Fig. 2. Upon receiving $observe_x$ from s_0 a tester cannot tell from $Spec_2$ if the communication is now at s_1 or at s_3. If it is at s_1 the SUT expects that the tester will send the next message ($control_a$), whereas if it is at s_3 the SUT expects that the tester will listen for the next message ($observe_y$). This dilemma is something that is not supposed to occur under synchronous communication, and the only

possible cause of it occurring is an imprecise specification that is given to a tester.

Upon encountering the dilemma, a tester might work around this issue by listening for an arbitrary amount of time to the SUT until it seems likely that the SUT will not send the next message, thus concluding that the SUT is probably in the act of listening to the tester. There are however a number of issues that arise from applying this workaround. First, we are putting the communication in a temporary hold status when the SUT is not in the act of sending which would be a waste of time. Second,

> ... always waiting for [the SUT] to respond before continuing can lead [the tester] to produce a test sequence that probably will not match the behavior of the real system. [6, p. 4]

And last, no matter for how long the tester decides to listen, when the tester does conclude the period of listening and sends some $control_$ message, there always remains a chance that the SUT is actually in the act of sending and not in the act of listening. When two parties are sending simultaneously under synchronous communication, it is no longer considered to be a valid communication sequence. A protocol can define what participants must do to recover from a situation in which multiple participants have sent messages simultaneously (e.g. restart from the initial state of the protocol). However, regardless of if and how this is explicitly specified, under normal circumstances a tester wants to maintain correct communication. There is no sound workaround to this issue. In related work authors prevent this by stating that their solution requires certain properties to hold on the LTS which together guarantee that this dilemma cannot occur.

It can easily be argued that putting these restrictions on specifications makes sense, as long as all sensible protocols can still be specified without violation of one of the restrictions. With synchronously communicated protocols this holds. However, the matter becomes much more complicated for asynchronously communicating protocols.

1.3 Testing Asynchronous Communication Protocols

In asynchronous communication the act of sending and receiving a message are two individual events. This means that participants do not synchronize every time communication takes place. Instead, a participant sends a message and is immediately free to continue its process. Meanwhile, the sent message is in transit and after some time arrives at its destination. Most often, participants have a receive buffer in which the arriving messages are stored FIFO until further processing. Compared to synchronous communication this method of communication is much more suited for communication between concurrent running systems when the cost of synchronization is large (e.g. when the duration of transit is large or can vary a lot). It is then the protocol's responsibility to introduce synchronization where required (e.g. hand shakes, or a directional synchronization: specifying that a participant must wait until it receives an acknowledgment message from the other participant).

Many service oriented protocols such as TCP/IP communicate asynchronously.

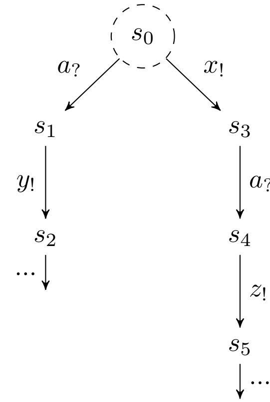


Fig. 3. Specification $Spec_3$.

Unfortunately, a consequence of this synchronize-less communication is that a tester can only directly infer the progress of the SUT from the $observe_$ messages it receives. A tester cannot know when exactly its $control_$ messages arrive at the SUT and neither can it know when exactly the SUT reads them from its receive buffer.

In [5], Tretmans formalizes the conformance relation under asynchronous communication between the specification and traces observed by a tester. He assumes that messages are sent through FIFO channels: one channel for sending messages to the SUT and one channel for receiving messages from the SUT. This work assumes the same. This means that at least the tester can assume that messages are not lost in transit and that messages sent through the same channel remain in order[1].

The delay in message transits along with the absence of synchronization enable, even under specification conforming communication, possible differences between the trace that was observed from the perspective of the tester and the trace that was observed from the perspective of the SUT. For example, consider $Spec_3$ shown in Fig. 3 and assume the tester sent $control_a$ from s_0. There exists a delay between this send event and the event wherein the SUT receives $control_a$. If the SUT sends $observe_x$ before it receives $control_a$, from the SUT's perspective the trace would be $x_! \cdot a_?$. Some time later the tester receives $observe_x$, and its observed trace of the communication becomes $a_? \cdot x_!$. Both participants were communicating according to the specification but nevertheless diverged in the traversal of the protocol.

The tester must consider the possible effects of asynchronicity when tracking in what possible states the SUT can be in. Tretmans defines the @ operator ($\sigma_1 @ \sigma_2$), read as: specification trace σ_1 could be aped (i.e. mimicked) by an observed trace σ_2. This operator captures exactly the specification traces that might have been taken from the SUT's perspective given a trace that was observed by the tester. To create some intuition some examples of traces that either are or are not aping some specification trace are

1. This assumption holds for all protocols that apply TCP for transmission of messages, or any other kind of transmission that guarantees that messages arrive in the same order they were sent. The assumption does not hold, for example, for protocols that apply UDP for transmission of messages.

Specification trace	Observed trace	Explanation
$x_!$ @	$a_? \cdot x_!$	$a_?$ could still be in transit.
$x_! \cdot a_?$ @	$a_? \cdot x_!$	The SUT might have sent $x_!$ prior to receiving $a_?$.
$b_?$ $\not@$	$a_? \cdot b_?$	$a_?$ must have reached the SUT prior to $b_?$, ruling out any specification trace that does not first consume $a_?$.
$a_? \cdot x_!$ $\not@$	$x_! \cdot a_?$	$a_?$ was sent after the tester received $x_!$. The SUT cannot have received $a_?$ after it sent $x_!$.
ϵ @	$a_? \cdot b_?$	Both $a_?$ and $b_?$ could still be in transit.
$a_?$ @	$a_? \cdot b_?$	$b_?$ could still be in transit.
$x_! \cdot b_?$ $\not@$	$a_? \cdot b_? \cdot x_!$	While $x_!$ may have been sent by the SUT before it received the control messages, $a_?$ must have been received prior to $b_?$.

Table 1
Some examples of specification traces that either have possibly been aped (i.e. mimicked) by an observed trace or have definitely not been aped by an observed trace. The observed traces are from the perspective of the tester.

shown in Table 1.

The ape operator is defined by Tretmans [5, p. 119] as follows.

- If $\sigma_1, \sigma_2 \in L^*_{ctrl}$, then $\sigma_1 @ \sigma_2 = \sigma_1 \preceq \sigma_2$
- If $\sigma_1 = ctrls_1 \cdot x1_! \cdot \sigma'_1, \sigma_2 = ctrls_2 \cdot x2_! \cdot \sigma'_2$, with $ctrls_1, ctrls_2 \in L^*_{ctrl}$, then $\sigma_1 @ \sigma_2 = ctrls_1 \preceq ctrls_2$ and $x1_! = x2_!$ and $\sigma'_1 @ (ctrls_2 \setminus ctrls_1) \cdot \sigma'_2$

The state that the tester is tracking becomes a set of possible states ($\mathbf{S}\{S^?_0, S^?_1, ..., S^?_n\}$). Each possible state $S^?$ is a pair (s_i, σ_{ctrl}), where s_i is a state of the specification ($s_i \in S$) and σ_{ctrl} is the list of $control_$ messages that are still either in transit or inside the SUT's receive buffer if the SUT is indeed at s_i. We refer to only the receive buffer of a possible state with $S^?_q$ (q here stands for *queue*), and to the LTS state field with $S^?_s$.

Whether a specification's state is reachable from some (possible) initial state (s_i, σ_{ctrl}) given a trace σ that was observed by the tester is now defined as follows.

$$(s_i, \epsilon) \xRightarrow[\text{Async}]{\sigma} (s_j, \sigma \setminus \sigma') \overset{\text{def}}{=} \sigma' @ \sigma \ \wedge \ s_i \xRightarrow[\text{Spec}]{\sigma'} s_j$$

$$(s_i, \sigma_c) \xRightarrow[\text{Async}]{\sigma} (s_j, \sigma'_c) \overset{\text{def}}{=} (s_i, \epsilon) \xRightarrow[\text{Async}]{\sigma_c \cdot \sigma} (s_j, \sigma'_c)$$

When determining the possibly reached states from some prior possible state (s_i, σ_{ctrl}), we must first prepend the control actions σ_{ctrl} (that were still in transit according that prior possible state) to the newly observed communication trace. Furthermore, all specification traces that may have been mimicked are considered by the application of the @ operator. The list of control messages that have not yet been consumed by a mimicked specification trace are stored in the possibly reached state's σ_{ctrl} field (this is described by $\sigma \setminus \sigma'$). The results of $\sigma \setminus \sigma'$ never contain any observe messages, since the traces that can be mimicked according the @ relation must always consume all observe messages.

For convenience, with the exception of formal definitions, a possible state that has no messages inside the receive buffer is often displayed with just the specification state id: (s_i **implies** (s_i, ϵ)).

Test strategies can be defined that try to achieve some predetermined goal through application of a set of test cases. A goal could for example be to perform all possible state transitions at least once. A test strategy determines at each step of the test process what the tester should do (e.g. send message $control_a$, or wait for incoming observe messages). These decisions are based on what continuation traces are allowed from each currently possible state.

$$async_traces(S^?) = \{\sigma \mid \exists S^{?\prime}.\ S^? \xRightarrow[\text{Async}]{\sigma} S^{?\prime}\}$$

A decision of a test strategy is **sound** if the action is allowed from every possible state in the current $\mathbf{S}\{..\}$[2]. For example, if the test strategy decides to send $control_a$ to the SUT, the decision is not sound assuming that this action is only allowed from some of the currently possible states in $\mathbf{S}\{..\}$. As a result, if it turned out that according to the SUT we were in one of the states where $control_a$ is not allowed to be sent (i.e. the first expected control action in all continuation traces is some other control message than $control_a$), we enter undefined behavior. The SUT might send an error message stating that it did not expect to receive $control_a$, or it might close the connection, or it might ignore the message. This depends on the protocol's procedure concerning reacting to unexpected message types. In any case, under normal procedure the test strategy should maintain correct communication and thus try to perform sound decisions. This work will not further address designing test strategies. It will however address how the tracking of states can be done such that the test strategy is supplied with unambiguous information concerning the current possible states (or at least as unambiguous as possible).

A function similar to $async_traces(S^?)$ is later employed by the state inference optimization technique (that will be explained in the following section) to consider the allowed continuation traces from the current state. It is important to notice that these traces are specification traces, where asynchronous effects are not considered.

$$spec_traces(s_i) = \{\sigma \mid \exists s_j.\ s_i \xRightarrow[\text{Spec}]{\sigma} s_j\}$$

The tracking of possible states given the observations of the tester can be performed following a number of different strategies.

I. Track in what states a correct SUT could be given the thus far observed trace. For example, with $Spec_3$ and an as of yet empty observed trace (ϵ) the SUT might have already sent, a still in transit message, $observe_x$. Thus under this tracking strategy the set of possible states with observed trace ϵ is: $\mathbf{S}\{s_0, s_3\}$. And considering an observed trace $a_?$ we would have $\mathbf{S}\{s_1, s_2, (s_0, a_?), (s_3, a_?), s_4, s_5, ...\}$. This tracking strategy clearly quickly becomes unmanageable.

II. Track in what states a correct SUT could be given the thus far observed trace, without considering possible in transit observe messages. These possibilities can safely be neglected because the @ operator catches situations where hindsight taught that an observe message was

2. With an addendum that sending a control action that results in a deadlock in the receiving channel of the SUT is not allowed either, even though the definition of $async_traces$ does not catch this.

Table 2
A communication trace observed by the tester of $Spec_3$ along with the set of states that the SUT might be in after each registered communication event. Empty $S_i^?$ cells indicate that the $\mathbf{S}\{..\}$ of the respective trace step contains less than $i+1$ possible states.

Step	Trace	$S_0^?$	$S_1^?$
0	ϵ	s_0	
1	$a_?$	$(s_0, a_?)$	s_1
2	$a_? \cdot x_!$	$(s_3, a_?)$	s_4
3	$a_? \cdot x_! \cdot z_!$	s_5	

in transit and subsequently correctly determines the mutations to the tracked state. For example, under this tracking strategy with $Spec_3$ and an observed trace ϵ we have: $\mathbf{S}\{s_0\}$. And with an observed trace $a_?$ we have $\mathbf{S}\{(s_0, a_?), s_1\}$. In the sequel, this strategy is what is referred to whenever we mention the tracking of the SUT's possible states. That is to say, when this phrase is mentioned we are not referring to the first strategy. The first strategy will never be mentioned again in this document.

III. Track in what states the *conversation* could be given a correct SUT. For now consider it to be equal to strategy II. In Section 2 an essential optimization on state inference is introduced that can only be applied to this tracking strategy.

This work applies strategy III. Again, for now this is semantically equal to strategy II. As an example consider Table 2 wherein, after each event that was registered by the tester, the states are listed where the SUT could be in.

The tracking strategy remains **sound** with regards to producing passing or failing verdicts when, at all time, one of the possible states in $\mathbf{S}\{..\}$ reflects the actual state of the communication (i.e. at least so long as the SUT communicates correctly). Assuming that the test strategy produces sound decisions, staying sound in the tracking strategy guarantees that a correct SUT will always pass any test case. In other words: the tester will never produce errors under correct behavior of both participants (i.e. the SUT and the tester's acting agent: the test strategy).

The final essential definition that Tretmans presents in his work is a computable function for determining which states are possible after a new observation. To clarify, the set of possible states $\mathbf{S}\{..\}$ is updated upon a new observation $a_$ by iteratively updating each prior possible state:

$$\bigcup_{(s_i, \sigma_c) \in \mathbf{S}\{..\}} \mu_{s_i}(\sigma_c \cdot a_)$$

Where μ_s is an inference algorithm that has been defined by Tretmans [5, p. 142]. Its definition is shown below.

$$\begin{aligned}
\mu_s(\epsilon) &= \{(s', \epsilon) \mid s \overset{\epsilon}{\underset{\text{Spec}}{\Longrightarrow}} s'\} \\
\mu_s(\sigma \cdot a_?) &= \{(s', \sigma' \cdot a_?) \mid (s', \sigma') \in \mu_s(\sigma)\} \\
&\quad \cup \{(s'', \epsilon) \mid (s', \epsilon) \in \mu_s(\sigma) \ \wedge \ s' \overset{a_?}{\underset{\text{Spec}}{\Longrightarrow}} s''\} \\
\mu_s(\sigma \cdot x_!) &= \{(s'', \sigma'') \mid \exists s', \sigma', \rho. \\
&\quad (s', \sigma') \in \mu_s(\sigma) \ \wedge \ \rho \ @ \ \sigma' \ \wedge \\
&\quad s' \overset{x_! \cdot \rho}{\underset{\text{Spec}}{\Longrightarrow}} s'' \ \wedge \ \sigma'' = \sigma' \setminus \rho\}
\end{aligned}$$

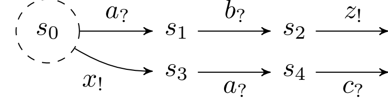


Fig. 4. Specification $Spec_4$.

Table 3
An observed communication trace of $Spec_4$.

Step	Trace	$S_0^?$	$S_1^?$
0	ϵ	s_0	
1	$a_?$	$(s_0, a_?)$	s_1
2	$a_? \cdot x_!$	$(s_3, a_?)$	s_4

This concludes the introduction of formalisms that have been acquired by related work. Everything hereafter has resulted from our own research and is based on the thus far introduced formalisms.

2 The Consequences of Asynchronicity

There are some issues that arise from tracking the possible states that the SUT could be in. This section introduces these issues, and tackles them through optimization of the tracking process. This section often considers inference steps taken following Tretmans' state inference algorithm (μ_s). This algorithm essentially considers if and how asynchronous effects have affected the state. We refer to applying μ_s with the keyword **async**. The optimization involves performing an inference step that disregards asynchronous effects. We refer to this with the keyword **sync**. It applies a state inference algorithm (ζ_s, Definition 1) that is actually meant to be used for synchronous communication tracking.

Definition 1.

$$\begin{aligned}
\zeta_s(\epsilon) &= \{(s', \epsilon) \mid s \overset{\epsilon}{\underset{\text{Spec}}{\Longrightarrow}} s'\} \\
\zeta_s(\sigma \cdot a_) &= \{(s'', \epsilon) \mid (s', \epsilon) \in \zeta_s(\sigma) \ \wedge \ s' \overset{a_}{\underset{\text{Spec}}{\Longrightarrow}} s''\}
\end{aligned}$$

2.1 Inference Inaccuracy Decreases Test Strategy Efficiency

Consider the observed communication trace shown in Table 3 according to the protocol specification $Spec_4$ shown in Fig. 4. The tester sent $control_a$ after step 0. **async** cannot know whether or not the SUT has received $control_a$ at step 1, and concludes that the SUT must be at either s_0 (with $control_a$ still in transit or in the SUT's input queue) or at s_1. At step 1 the test strategy has to deal with this inaccurate information. Assuming that the SUT is at s_1 the communication will not continue until the tester sends $control_b$. However, assuming that the SUT is still at s_0, the SUT might decide to send $observe_x$. Step 2 shows that the latter indeed happened. **async** infers at this point that the SUT must be either at s_3 (with $control_a$ still in its input queue) or at s_4. Under the new $\mathbf{S}\{..\}$ at step 2, the tester is not allowed to send $control_b$ at all.

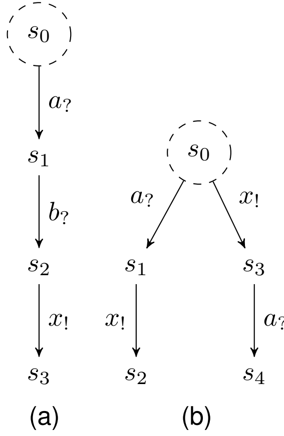


Fig. 5. Specifications $Spec_{5A}$ and $Spec_{5B}$ are shown in (a) and (b) respectively.

Table 4
An observed communication trace of $Spec_{5A}$.

Step	Trace	$S_0^?$	$S_1^?$	$S_2^?$
0	ϵ	s_0		
1	$a_?$	$(s_0, a_?)$	s_1	
2	$a_? \cdot b_?$	$(s_0, a_? \cdot b_?)$	$(s_1, b_?)$	s_2
3	$a_? \cdot b_? \cdot x_!$	s_3		

In more aggressive approach, the tester could have sent $control_b$ immediately at step 1. If $observe_x$ was already in transit, doing so would result in $a_? \cdot b_? \cdot x_!$ from the tester's perspective, which is not an observable trace of $Spec_4$. Essentially, not taking the accuracy of the information along with the asynchronicity into account can result in observing incorrect SUT behavior from the tester's perspective while from the SUT's perspective it is the tester that is behaving incorrectly (from the SUT's perspective the trace would be $x_! \cdot a_? \cdot b_?$). At step 1 the tester has to decide for how long it will wait until it considers the odds of the SUT sending $observe_x$ low enough to challenge the risk of bad timing and to send $control_b$. Obviously this predicament should be prevented if possible. To do so it is key that **S**$\{..\}$ is kept as accurate as possible. However, inaccuracy is clearly not always preventable (case in point Table 3).

As the number of possible states in **S**$\{..\}$ grows, **S**$\{..\}$ reflects the actual conversation state less accurately. Under low accuracy, the test strategy has to make decisions based on low certainty information. Losing soundness is unacceptable and losing accuracy increases the difficulty to effectively apply a test strategy. It is thus important to both remain sound and remain as accurate as possible.

2.2 Async (μ_s) Increases Inaccuracy

Table 4 shows an observed communication trace of $Spec_{5A}$ (Fig. 5). At step 1, **async** takes in consideration that the SUT has not yet received $control_a$. Similarly, at step 2 **async** takes in consideration that perhaps the SUT has not yet received both control messages, or that it perhaps has received $control_a$ by now but not yet $control_b$. However, this does not matter from the conversation viewpoint. There are no possibly correct asynchronous effects since the SUT is not allowed to send messages at this stage. The conversation must thus be at s_1 after the tester sent $control_a$, and it must be at s_2 after the tester sent $control_b$.

Conversely, $Spec_{5B}$ in Fig. 5 shows a case where it is required to consider the asynchronous nature of the communication method. If the tester sends $control_a$ when we are at s_0, it is not yet sure if the SUT sent $observe_x$ before receiving $control_a$ or if it will send it after $control_a$. Depending on this order the conversation either reaches s_2 or s_4, and the tester must consider both possibilities.

Considering these cases, it is desired to statically determine (from the specification) when it is required to consider the asynchronous effects and when it is safe (i.e. we remain sound w.r.t. tracking the conversation state) to disregard these effects. Theorem 1 gives this relation. In the sequel every inference step is either applied using **async** (which takes asynchronous effects into consideration) or using **sync** (which disregards asynchronous effects). When not otherwise stated, **async** is applied by default.

Theorem 1. When updating some $S^?$ upon sending a message $control_a$, **sync** finds sound results iff

$$S_q^? = \epsilon \wedge$$
$$\forall \sigma.\ \sigma \in spec_traces(S_s^?) \ \wedge\ \sigma @ a_? \cdot * \textbf{ we have } \sigma = a_? \cdot *$$

If $control_a$ is the first event of all the specification's continuation traces, then there exist no asynchronous effects through which the SUT might end up at a different state in the specification. Thus, it is safe to update the state as if the $control_a$ was communicated synchronously.

The continuation traces list as well as each of its elements are unbounded in size (up to infinity given that there are loops present in the protocol). However, it is only necessary to consider the first few actions of a finite subset of the traces to either prove or disprove the applicability of **sync** following Theorem 1.

- Tretmans found the following relation between a specification and the observed traces that it allows.

  > Given $\sigma_0 @ \sigma_1$, the causal dependencies of control actions on previous control actions in σ_0 must be preserved in σ_1. [5, p. 120]

  From this relation follows:

  $$\forall b_?.\ b_? \neq a_? \textbf{ implies } observe_-^* \cdot b_? \cdot * \not@\ a_? \cdot *$$

  Thus, any $\sigma \in spec_traces(S^?)$ with $\sigma = observe_-^* \cdot b_? \cdot *$ and $b_? \neq a_?$ immediately does not satisfy the quantification's domain in Theorem 1, and hence is not considered. Any trace that does not have $control_a$ as its first control action does not contribute to proving nor disproving Theorem 1, and trivially any trace that does have $control_a$ as its first control action immediately either proves or disproves that this trace satisfies Theorem 1. From this it follows that any trace only has to be considered up to its first control action.
- The traversal of loop structures only has to be considered in the range of 0 to 1 iterations. Either:
  - its first control action is $control_a$, proving or disproving that this trace satisfies Theorem 1.

$mustAsyncs = []$
for all $s \in Places$ **do**
$afterObs = \{e.dest | e \in Edges \wedge e.org = s \wedge e.label = observe_\}$
$asyncEvents = \bigcup_{s' \in afterObs} helper\ \emptyset\ s'$
$mustAsyncs = (s, asyncEvents) : mustAsyncs$
end for

function $helper(ss, s)$ ▷ ss is the set of visited states
$ss' = \{s\} \cup ss$
$afterObs = \{e.dest | e \in Edges \wedge e.org = s \wedge e.label = observe_\} \setminus ss'$
$ctrlEvs = \{e.label | e \in Edges \wedge e.org = s\}$
return $ctrlEvs \cup \bigcup_{s' \in afterObs} helper\ ss'\ s'$
end function

Fig. 6. The algorithm for determining from each state after which control events **async** must be applied.

- its first control action is some action other than $control_a$, disqualifying the trace since it does not satisfy the quantification in Theorem 1.
- The loop consists of only observe actions, in which case it must be considered that anything following the loop followed either 0 iterations of the loop (which considers a trace without observe actions originating from the loop) or at least 1 iteration of the loop (which considers some observe actions originating from the loop).

This bounds the size of the subset of traces as well as the length of each trace that needs to be considered.

Applying **async** after a control action increases the number of possible states. Assuming that it produces sound results, **sync** produces a more accurate set of possible states than **async** does. Theorem 2 presents the same relation as Theorem 1, but formulates it the other way around (i.e. it searches for proof that **async** must be applied, instead of searching for proof that **sync** cannot be applied). In the sequel we will use this format to prove or disprove applicability of this optimization given some situation. Fig. 6 presents the algorithm that, given the specification, statically determines from each state after which control actions we must apply **async** to stay sound. This algorithm approaches the challenge from a similar direction as Theorem 2 does.

Theorem 2. To stay sound after sending $control_a$, **async** *must* be applied to update $S^?$ iff

$$S^?_q \neq \epsilon\ \vee \quad (1)$$

$$\exists s_j.\ S^?_s \xrightarrow[\text{Spec}]{observe^+_{_} \cdot a_?} s_j \quad (2)$$

Where $observe^+_{_}$ is some non-empty sequence of observe actions:

$$observe^+_{_} = \exists x, \sigma_{obs}.\ x_! \cdot \sigma_{obs}$$

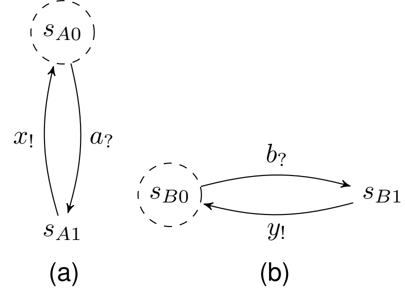


Fig. 7. Specification segments $Spec_{6A}$ and $Spec_{6B}$ are shown in (a) and (b) respectively.

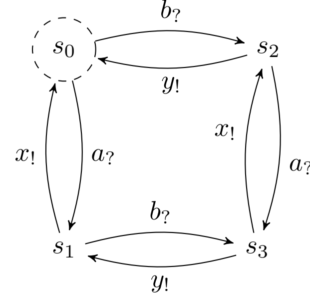


Fig. 8. Specification $Spec_6$. $Spec_6$ was formed by combining specification segments $Spec_{6A}$ and $Spec_{6B}$.

2.3 Parallel Segments Increase Inaccuracy

A protocol can define communication segments that are traversed in parallel. Essentially, there are no restrictions to the composition of these segments, each segment describes a communication sequence whose traversal does not affect the other segment's traversals. Consider a protocol where at some point in the communication both segments displayed in Fig. 7 (segment $Spec_{6A}$ and $Spec_{6B}$ respectively) are traversable in parallel. Regardless of where the communication trace is in $Spec_{6B}$, if $Spec_{6A}$ is at state s_{A0}, $control_a$ can be sent by the tester. Similarly, if $Spec_{6B}$ is at state s_{B1}, the SUT is allowed to send $observe_y$ regardless of in what state $Spec_{6A}$ is in.

It is possible to model this phenomenon with a single LTS. In Fig. 8 $Spec_6$ is shown which combines the parallel communicated specifications $Spec_{6A}$ and $Spec_{6B}$. We leave it to the reader to convince himself/herself that $Spec_6$ indeed allows the parallel segments to be traversed independently.

Suppose that we are communicating purely according to $Spec_{6A}$ and we have $\mathbf{S}\{s_{A0}\}$ and $control_a$ is sent. According to Theorem 2 we remain sound if we apply **sync**. As a result, we end up with $\mathbf{S}\{s_{A1}\}$ (whereas applying **async** would give $\mathbf{S}\{(s_{A0}, a_?), s_{A1}\}$). Similarly, suppose that we are communicating purely according to $Spec_{6B}$ and we have $\mathbf{S}\{s_{B0}\}$ and $control_b$ is sent. Again we can remain sound if we apply **sync**. Doing so results in $\mathbf{S}\{s_{B1}\}$.

Consider a similar communication pattern given $Spec_6$ where, after a certain trace, we have $\mathbf{S}\{s_1\}$ and $control_b$ is sent. Since we have $s_1 \xrightarrow[\text{Spec}]{x_! \cdot b_?} s_2$, Theorem 2 cannot guarantee soundness if we apply **sync**. To remain sound **async** must be applied, resulting in $\mathbf{S}\{(s_1, b_?), s_3\}$. Ap-

Table 5
An observed communication trace of $Spec_{5A}$. Possible states that were determined using **sync** are underlined within the **S**{..} notation.

Step	Trace	$S_0^?$
0	ϵ	s_0
1	$a_?$	$\underline{s_1}$
2	$a_? \cdot z_!$	

parently, combining parallel communicated segments together into a single LTS loses information resulting in a less effective application of Theorem 2. A careful reader might have noticed that $Spec_6$ is commutative. As such, semantically (w.r.t. what actions are allowed and are not allowed at each intermediate **S**{..}) it does not matter if we traverse under synchronized communication from s_1 to s_2 (which in this case would always traverse via s_3) or under asynchronized communication from s_1 to s_2 (which would consider both the traverses via s_0 and s_3). Due to the commutative property of combined parallel communicated segments this always holds. From this follows that (at least seemingly so) it should be possible to expand Theorem 1 and Theorem 2 such that they are able to statically determine the commutative nature of a current part of the protocol.

However, it is not clear what the impact is of synchronization patterns on the capability of statically determining commutative parts. A synchronization pattern is for example a segment waiting for another segment to reach a certain state, or for example the conclusion of parallel segments and only when all segments have reached one of their respective final states will the protocol continue.

Besides the uncertainty following from the aforementioned issue, there are other reasons why it is favorable to instead prevent this issue entirely by using another paradigm than Labeled Transition Systems to describe the protocol. Petri Nets offer the abstractions that we are looking for and will be further discussed in Section 3.

2.4 Sound Conversation States cannot Produce Sound Error Reports

Unfortunately, there are downsides to tracking the conversation state as opposed to the SUT state. Table 5 shows an observed communication trace of the specification $Spec_{5A}$ shown in Fig. 5. At step 2 the tester received $observe_z$ and correctly concluded that this action was not allowed according to $Spec_{5A}$. Subsequently, the tester proceeds by reporting the error and operators will have to decipher what went wrong.

If we had been tracking the SUT's state, **async** would have been applied at each inference step, resulting in the last valid state (at step 1): **S**{(s_0, $a_?$), s_1)}. We were however tracking the conversation state. As such, it was valid to apply **sync** to determine the state at step 1 (which is what happened as shown in the communication trace).

The SUT either sent $observe_z$ before or after it received $control_a$. An error report solely based on the last valid *SUT state* at step 1 successfully accounts for both the former and the latter case. However, an error report that is solely based on the last valid *communication state* at step 1 fails to account for the former case.

Since tracking SUT states stays sound w.r.t. where the SUT is after each event, we can always produce sound error reports from the last determined valid **S**{..}. However, tracking the conversation state loses this property and cannot produce sound error reports from the last determined valid **S**{..}.

It might however be possible (or even trivial) to obtain the last valid SUT state by reasoning asynchronous effects backwards from the last valid conversation state along with the observed communication trace. At the least there is a naive solution available, which is to simply reconsider the entire trace only this time tracking the SUT states instead of the conversation states.

3 A Sound and Accurate APSL Engine

In [7] the authors frame an essential difference between testing a sequential program (to which synchronously communicating systems can be directly related) and testing concurrent systems.

> The semantics of a concurrent system can neither be simply defined nor adequately tested as a partial function from inputs to outputs although it is appropriate for a sequential program.

This is then followed up with the essential conclusion:

> Instead, it must be *defined* in terms of its *dynamic behavior* and tested by observing the dynamic behavior.

We agree with this viewpoint.

Similar to them (whom were researching white box testing of concurrent systems) we propose to use Petri Nets to formally model protocols. We can subsequently base MBT on the Petri Net specification. Petri Nets do not model a global state directly (whereas an LTS does) but instead model the transitions of parts of the global state as dynamic behavior.

We opted to use Petri Nets, because they can describe parallel behavior explicitly and do not (seem to) impose limits on what can and cannot be modeled. It may be that there exist other modeling paradigms which also allow to describe parallel behavior and are better suited as a basis for MBT of protocol implementations. For example, it is quite complex to compute the continuation traces from a current "state" (i.e. the $spec_traces(s_i)$ equivalence under the Petri Nets paradigm, this will be explained later in this section), which may increase difficulty to design effective test strategies. This is an interesting topic that can be addressed by future work.

3.1 Petri Nets

The descriptive power of Petri Nets [8] differs from that of Finite State Machines. A Petri Net (PN) consists of a set of *places* ($p \in P$) and a set of *transitions* ($t \in T$). The state that a PN is in (i.e. the *marking* or current *configuration* of a PN, by PN terms) is described by a multiset (bag) of *markers*, each residing at one of the places. A transition ($srcs \rightarrow dests$) is described with a set of incoming edges from places (the transition's sources: $srcs \subseteq P$) and a set of outgoing edges to places (the transition's destinations: $dests \subseteq P$). A transition can be *fired* when at all of its

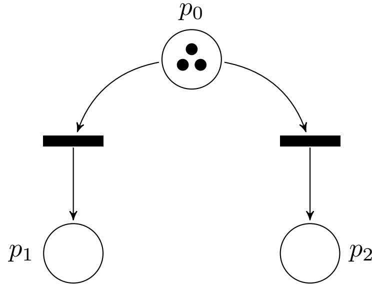


Fig. 9. The visualization of a PN's marking. Places are visualized with circles, transitions are visualized with horizontal or vertical bars and each dot at a place represents a marker. In this marking, there are 3 markers at p_0 and 0 markers at both p_1 and p_2.

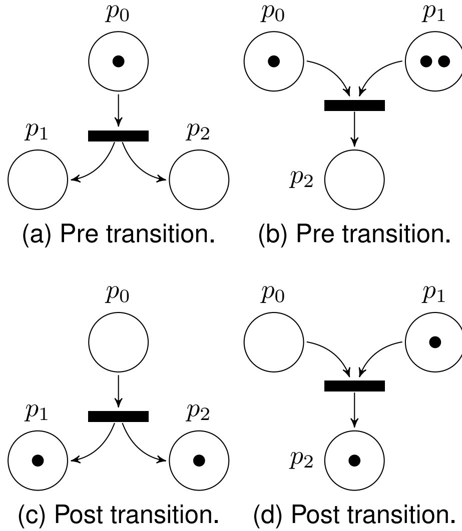


Fig. 10. The marking before (a, b) and after (c, d) a fired transition. The transition in (a) has multiple outgoing edges, resulting in a marker at each destination (p_1 and p_2) in (c). The transition in (b) has multiple incoming edges, resulting in the removal of a marker at each source (p_0 and p_1).

sources at least 1 marker is present. Upon firing a transition, a marker at each of its sources is removed and a marker at each of its destinations is created. In Fig. 9 a marking of a simple PN is visualized. There are 3 markers at p_0. Firing either transition will reduce the number of markers by 1 at p_0, and depending on which transition was fired increase the markers at either p_1 or p_2 by 1.

The difference in descriptiveness of PNs compared to FSMs stems from its transitions and how they operate on markers. An edge in a FSM can be compared to a transition in a PN that has a single source and a single destination. In fact, any FSM can be represented by a PN by replacing each edge $s_i \rightarrow s_j$ with a transition $[s_i] \rightarrow [s_j]$. In this PN, assuming we start with a single marker, the marking will always consist of a single marker, which is essentially what all FSMs model (a FSM is always in a single state). However, a transition with multiple sources or multiple destinations introduces new semantics. This allows for explicitly describing the entering of parallel segments as well as the synchronization of parallel segments.

Fig. 10 shows the effect of firing a transition that has

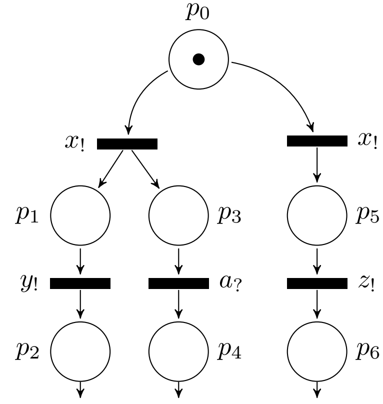


Fig. 11. Specification $Spec_7$.

Table 6
An observed communication trace of $Spec_7$.

Step	**Trace**	$C_0^?$	$C_1^?$
0	ϵ	$[p_0]$	
1	$x_!$	$[p_5]$	$[p_1, p_3]$
2	$x_! \cdot y_!$	$[p_2, p_3]$	

multiple outgoing edges and of firing a transition that has multiple incoming edges. The transition at (b) is only firable when at all of its sources there is a marker present.

It is important to realize the difference between multiple attached edges to a place (Fig. 9) and multiple attached edges to a transition (Fig. 10). Where multiple outgoing edges from a *transition* denotes the duplication of markers, multiple outgoing edges from a *place* represents a choice (a marker can only be consumed by a single attached transition). And where multiple incoming edges to a transition denotes the merging of markers, multiple incoming edges to a place simply means that a marker can arrive at this place via different routes.

In the case of APSL we can use the markers to track where we are in the communication. We had to extend the notion of FSM to LTS and extend the state of the LTS to a set of possible states, we have to apply these same measures for similar reasons to our PN formalism. The transitions are labeled with message events (or the absence of a visible event: τ), and can now only be fired upon registering the occurrence of the event (in addition to the fact that each source place must have at least one marker present). And the state of the communication is described not by a single configuration, but by a set of possible configurations (**C**$\{C_0^?, C_1^?, .., C_n^?\}$). Each possible configuration $C^?$ describes a marking with a bag of markers $m(\forall m \in M : m \in P)$: $C^?[m_0, m_1, .., m_n]$. $C^?$ additionally contains the state of the SUT's input queue: $C^?([m_0, m_1, .., m_n], a_? \cdot b_? \cdot *)$. We refer to only the receive buffer of a possible configuration with $C_q^?$, and to its marking with $C_M^?$. The set of possible configurations (**C**$\{..\}$) actually describes in what state the system is of a subset construction applied to the PN.

As a demonstration consider specification $Spec_7$ (Fig. 11) and the communication trace shown in Table 6. The tester received $observe_x$ from the SUT from p_0. $Spec_7$ has

two transitions that can fire, but only 1 is allowed to fire (notice: not only because there are not enough markers at p_0, but also because this would require receiving $observe_x$ twice). At this point the tester cannot know which of the transitions was traversed by the SUT and for now considers both, concluding that at step 1 the communication could either be at $C^?_0[p_5]$ or at $C^?_1[p_1, p_3]$. Notice that $C^?_1$ has two markers, one at p_1 and the other at p_3. Thus, assuming $C^?_1$ is the actual communication's configuration, from here on the communication is allowed to traverse further in parallel from two markers. After step 1 of the communication trace the tester received $observe_y$. $C^?_0$ in step 1 has no markers that can traverse, disqualifying this marking from the set of possible configurations. $C^?_1$ in step 1 has one marker (at p_1) that can traverse on receiving $observe_y$. This results in the $\mathbf{C}\{..\}$ at step 2 that has only a single possible configuration with two markers, one at p_2 and the other at p_3.

Consider a similar protocol as $Spec_7$, instead now with the transition from p_3 to p_4 labeled with the same event as the transition from p_1 to p_2: $observe_y$. With this specification and an observed communication trace $x_! \cdot y_!$, the tester cannot know if the marker at p_1 traversed to p_2 or if the marker at p_3 traversed to p_4. Although the markers are both allowed to traverse forward on this event, only exactly one of the transitions can fire on receiving $observe_y$. Thus, the tester has to consider that either the one or the other fired, resulting in again two possible configurations: $\mathbf{C}\{C^?[p_2, p_3], C^?[p_1, p_4]\}$. Similar to when we were using an LTS to track the communication state, non deterministic behavior can arise from transitions that fire on the same event. However, unlike with the LTS paradigm it is not necessary that, without asynchronous effects taking place, these transitions are attached to equal sources.

In the LTS paradigm we used $s_i \xrightarrow{a} s_j$ to describe that we can traverse from s_i to s_j on event a. This traversal would conform to the specification iff $(s_i, a, s_j) \in E$. A transition in a PN can capture more complex prerequisites and modifications to the global state (notice: the marking defines the state of the system), hence relating a similar description of going from some initial marking M to a new marking M' on event a ($M \xrightarrow{a} M'$) to the specification involves some computation. Definition 2 formally defines the relation.

Definition 2.

$$M \xrightarrow[\text{Spec.}]{a} M' \stackrel{\text{def}}{=}$$
$$M' \in \{(M \setminus t.srcs) \cup t.dests \mid \exists t.\ t \in T: t.ev = a \land t.srcs \subseteq M\}$$

This is the only operator that differs from the LTS variant. The following essentially remain unmodified. Except, they do apply the modified $.. \xrightarrow[\text{Spec}]{a} ..$ PN variant internally instead of the LTS variant.

$$M \xRightarrow[\text{Spec.}]{\sigma} M' = s_i \xRightarrow[\text{Spec.}]{\sigma} s_j$$
$$C^? \xRightarrow[\text{Async.}]{\sigma} C^{?\prime} = S^? \xRightarrow[\text{Async.}]{\sigma} S^{?\prime}$$
$$\mu_M(\sigma) = \mu_s(\sigma)$$

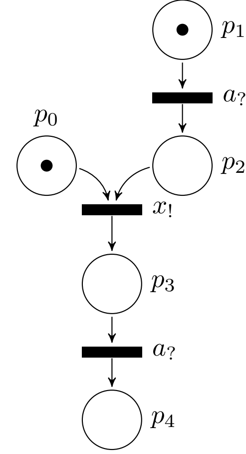


Fig. 12. Specification $Spec_8$.

$spec_traces(s_i)$ can be defined similarly for the PN formalism such that it determines the continuation traces from some current possible marking.

$$spec_traces(M) = \{\sigma \mid \exists M'.\ M \xRightarrow[\text{Spec}]{\sigma} M'\}$$

However, it is not clear if this function is computable for finite lengths of traces. Consider for example specification $Spec_8$ in Fig. 12, which has a current configuration that presents a sequence wherein two markers will synchronize (by the transition on $observe_x$). Synchronizations of markers must be considered correctly according to the definition of $spec_traces(M)$, consequently the order of firing transitions matters when there are multiple markers present. Even though there is a marker at p_0 (which is one of the sources of the transition towards p_3), $x_!$ is not an element of $spec_traces([p_0, p_1])$, $a_? \cdot x_!$ on the other hand is. According to the specification, before $observe_x$ is allowed, the marker at p_1 must first arrive at p_2. Although this example does not impose real challenges on computing the continuation traces following the definition of $spec_traces(M)$, it is not clear how to compute traces given a specification that intermingles loops and synchronizations of markers. In this sense, the challenge of computing $spec_traces(M)$ seems to be equivalent to the (probably more general) reachability problem of Petri Nets. If it is then $spec_traces(M)$ can be computed by applying well known solutions of this reachability problem [9].

This has not been further researched by this work. It is a complex problem, and solving it less precisely did not seem to have significant implications on the results of this work (it may however impose significant issues on results of future work). Thus, we approached the computation of possibly allowed traces naively.

$$\widetilde{spec_traces}(M) = \bigcup_{p \in M} \{\sigma \mid \exists p'.\ p \underset{\text{Spec}}{\overset{\sigma}{\approx>}} p'\}$$

Here, $\underset{\text{Spec}}{\approx>}$ over-approximates what traces are allowed by ignoring the synchronization of markers (its definition is shown below). $\widetilde{spec_traces}(M)$ can thus ignore the marking of the net. This greatly simplifies the computation, making

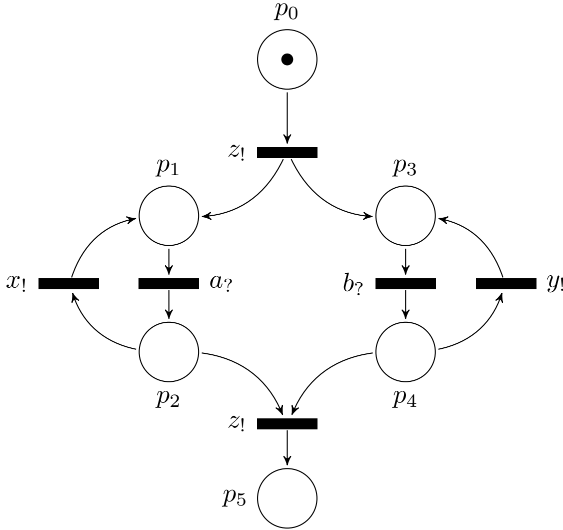


Fig. 13. Specification $Spec_9$.

Table 7
An observed communication trace of $Spec_9$. Where it was safe to apply **sync** after observing a $control_$ event (according Theorem 3) the resulting updated marker is underlined in the respective $C^?$.

Step	Trace	$C_0^?$
0	ϵ	$[p_0]$
1	$z_!$	$[p_1, p_3]$
2	$z_! \cdot a_?$	$[\underline{p_2}, p_3]$
3	$z_! \cdot a_? \cdot b_?$	$[p_2, \underline{p_4}]$
4	$z_! \cdot a_? \cdot b_? \cdot z_!$	$[p_5]$

it a graph walk that can be individually considered per marker.

$$p_i \underset{\text{Spec}}{\overset{a}{\rightsquigarrow}} p_j \overset{\textbf{def}}{=} \exists t.\ t \in T \ \wedge$$
$$t.ev = a \ \wedge \ p_i \in t.srcs \ \wedge \ p_j \in t.dests$$
$$p_i (\underset{\text{Spec}}{\overset{a}{\rightsquigarrow}})^+ p_j \overset{\textbf{def}}{=} p_i \underset{\text{Spec}}{\overset{a}{\rightsquigarrow}} p_j \ \vee \ \exists p_i'.\ p_i (\underset{\text{Spec}}{\overset{a}{\rightsquigarrow}})^+ p_i' \underset{\text{Spec}}{\overset{a}{\rightsquigarrow}} p_j$$

$$p_i \underset{\text{Spec}}{\overset{\epsilon}{\approx\!>}} p_j \overset{\textbf{def}}{=} p_j = p_i \ \vee \ p_i (\underset{\text{Spec}}{\overset{\tau}{\rightsquigarrow}})^+ p_j$$
$$p_i \underset{\text{Spec}}{\overset{a \in L}{\approx\!>}} p_j \overset{\textbf{def}}{=} \exists p_i',\ p_i''.\ p_i \underset{\text{Spec}}{\overset{\epsilon}{\approx\!>}} p_i' \underset{\text{Spec}}{\overset{a}{\rightsquigarrow}} p_i'' \underset{\text{Spec}}{\overset{\epsilon}{\approx\!>}} p_j$$
$$p_i \underset{\text{Spec}}{\overset{a_0 \cdot a_1 \cdot \ldots \cdot a_n \in L^*}{\approx\!>}} p_j \overset{\textbf{def}}{=} \exists p_i'.\ p_i \underset{\text{Spec}}{\overset{a_0}{\approx\!>}} p_i' \underset{\text{Spec}}{\overset{a_1 \cdot \ldots \cdot a_n}{\approx\!>}} p_j$$

Theorem 1 implies that sound application of **sync** can be decided by verifying that every allowed continuation trace according to the protocol from a current $S^?$ is not disregarded when ignoring asynchronous effects. The same theorem still holds with PN specified protocols. In fact, the markers allow for a more accurate notion of allowed traces.

Theorem 3 presents a PN paradigm version of Theorem 1. It achieves an increased accuracy by only considering asynchronous effects within each parallel segment (at least, to a limited degree due to the naive traces computation).

Theorem 3. When updating some $C^?$ upon sending a message $control_a$, **sync** finds sound results if (but not only if, due to our over approximated continuation traces computation)

$$C_q^? = \epsilon \ \wedge$$
$$\forall \sigma.\ \sigma \in \widetilde{spec_traces}(C_M^?) \ \wedge \ \sigma @ a_? \cdot * \ \textbf{we have } \sigma = a_? \cdot *$$

3.2 Parallel Segments Revisited

Fig. 13 presents specification $Spec_9$, a protocol similar to $Spec_6$. Only now the protocol is described with a PN instead of an LTS and has been extended with an initial traversal from p_0 into the two parallel communicated segments plus the synchronized finalization of the segments towards the final place of the protocol (p_5).

In Section 2 we discussed the issues that arise when the communication state is tracked using an LTS that implicitly contains multiple parallel communicated segments. Information that is required to improve the accuracy of the conversation state tracker is lost in the process of combining these sections together into a single LTS. Table 7 shows a communication trace of PN specification $Spec_9$. The LTS variant essentially failed to prove that it is sound to apply **sync** upon registering event $control_b$ when in a combined marker state of p_2 and p_3. The table shows at step 3 that when using the PN to track the communication, under similar pre conditions, it is able to prove that it can apply **sync** safely.

3.3 θ transitions

The EDSL that will be introduced in the following section (Section 4) generates in many of its instructions transitions that can be fired without the registration of an event. The "event" is dubbed θ, and is semantically in many ways comparable to τ "events". However, there is a subtle difference in considering when they are allowed to fire. The state tracker is forced to consider that τ transitions that are firable have potentially fired from the SUT's perspective. θ transitions on the other hand are applied by our EDSL to represent that the state is *allowed* to progress silently (i.e. without registering either an observe or control action) and is essentially both before and behind the transition simultaneously. There are two distinctive strategies concerning the firing of θ transitions.

The first fires θ transitions as soon as possible. We call this the greedy strategy. Table 8 demonstrates the effect of greedily firing θ transitions. After receiving $observe_x$ the state tracker immediately takes into consideration that the θ transition from p_1 to p_2 may fire. This results in a $\mathbf{C}\{..\}$ at step 1 that consists of two possible configurations: $C_0^?[p_1]$ and $C_1^?[p_2]$. The test strategy subsequently decides what the tester should do next. It has the specification and the possible continuation traces of each $C^?$ to base this decision on:

$$\widetilde{spec_traces}(C_0^?) = \{a_?,\ a_? \cdot b_?,\ b_?\}$$
$$\widetilde{spec_traces}(C_1^?) = \{b_?\}$$

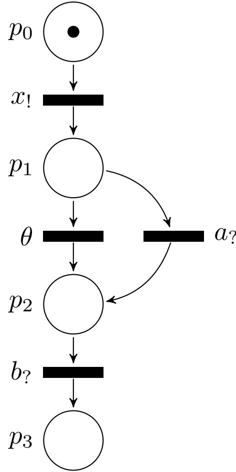


Fig. 14. Specification $Spec_{10}$.

Table 8
An observed communication trace of $Spec_{10}$ when greedily firing θ transitions.

Step	Trace	$C_0^?$	$C_1^?$
0	ϵ	$[p_0]$	
1	$x_!$	$[p_1]$	$[p_2]$
2	$x_! \cdot b_?$	$[p_3]$	

If only basing its decision on the possible continuation traces and not doing this in conjunction with some closer analysis of the specification (in this case noticing that $C_1^?$ is reachable from $C_0^?$ through the θ transition), the test strategy will always opt for sending $control_b$ since this action does not violate a continuation of any of the possible configurations. This would unfortunately result in failing to ever test the firing of the transition on $control_a$.

The alternative strategy is to fire θ transitions as late as possible. We call this the lazy strategy. Table 9 considers the same communication trace as Table 8 does only now under application of the lazy θ firing strategy. This strategy results in a more accurate $\mathbf{C}\{..\}$ at step 1, which subsequently results in unequivocal continuation information:

$$\widetilde{spec_traces}(C_0^?) = \{a_? \\ , a_? \cdot b_? \\ , b_?\}$$

Because the original definition of the reachability operator ($\Rightarrow$) greedily fires τ transitions, we should distinguish the firing of θ transitions from how we consider the firing of τ transitions. Definition 3 redefines the reachability operator

Table 9
An observed communication trace of $Spec_{10}$ when lazily firing θ transitions.

Step	Trace	$C_0^?$
0	ϵ	$[p_0]$
1	$x_!$	$[p_1]$
2	$x_! \cdot b_?$	$[p_3]$

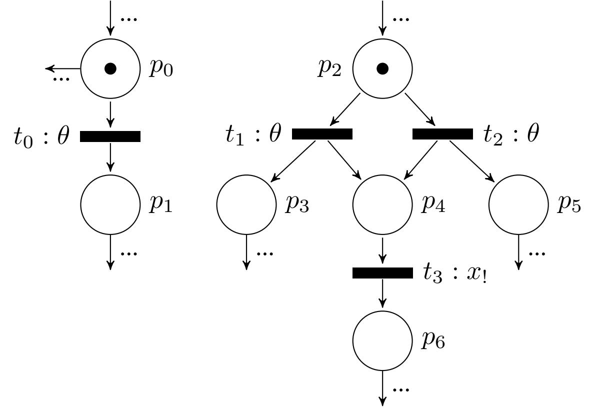


Fig. 15. Specification $Spec_{11}$.

Table 10
An observed communication trace of $Spec_{11}$ under lazily firing θ transitions.

Step	Trace	$C_0^?$	$C_1^?$
0	ϵ	$[p_0, p_2]$	
1	$x_!$	$[p_0, p_3, p_6]$	$[p_0, p_5, p_6]$

to lazily fire θ transitions. There are some notations applied that have not been formally introduced. The t captures the transition that is fired on the a event (.. $\xrightarrow[\text{Spec}, t \in T]{a} M'$). The ts set captures only the θ transitions that are on a marker path of M to one of the source places of t. And $(\xrightarrow{\theta})_{ts}$ only considers the firing of a θ transition that is in ts.

Definition 3. The reachability operator's single step definition under the lazy θ transition firing strategy.

$$M(\xrightarrow[\text{Lazy}]{\theta})^*_{ts} M' \overset{\text{def}}{=} \\ M' = M \lor \\ \exists M''.\ M(\xrightarrow[\text{Lazy}]{\theta})_{ts} M''(\xrightarrow[\text{Lazy}]{\theta})^*_{ts} M'$$

$$M \xrightarrow[\text{Lazy}]{a \in L \,\lor\, \tau} M' \overset{\text{def}}{=} \\ \exists M'',\ ts = \{t' \mid \exists m, p. \\ t' \in T \ \land\ m \in M \ \land\ p \in t.srcs \land \\ t' \in m(\underset{\text{Spec}}{\overset{\theta}{\rightsquigarrow}})^+ p\}. \\ M(\xrightarrow[\text{Lazy}]{\theta})^*_{ts} M'' \xrightarrow[\text{Spec}, t \in T]{a} M'$$

The following essentially remain unmodified. Except, they do apply the modified .. $\xrightarrow[\text{Lazy}]{a}$ .. variant internally instead of the Spec variant.

$$M(\xrightarrow[\text{Lazy}]{a})^+ M' \overset{\text{def}}{=} M(\xrightarrow[\text{Spec}]{a})^+ M'$$

$$M \overset{\sigma}{\underset{\text{Lazy}}{\Longrightarrow}} M' \overset{\text{def}}{=} M \overset{\sigma}{\underset{\text{Spec}}{\Longrightarrow}} M'$$

Considering the firing of θ transitions that cannot possibly contribute to enabling the firing of a transition $a \in L$ is greedy and results in unnecessary growth of the set of

possible configurations. This is prevented by only considering firing θ transitions that are on a marker path towards a reachable $a \in L$ transition.

For example, consider Fig. 15 and Table 10. The tester received $observe_x$. Assuming that there is no transition adjacent to p_0 or p_1 that consumes this message, t_0 must not be considered in the $(\xrightarrow{\theta})^*_{ts}$ of Definition 3. And it isn't, because it is not in a marker path from M (where $M = [p_0, p_2]$) to a source place of the only reachable $observe_x$ transition t_3. Furthermore, transition t_3 can consume this event by first firing either θ transition t_1 or θ transition t_2. It is important that both are considered because θ transitions can produce side effects (i.e. they can multiply and merge markers). In this example, firing t_1 followed by t_3 effectively removes the marker at p_2 and creates a marker at p_3 and p_6, whereas firing t_2 followed by t_3 does not create a marker at p_3 and instead creates a marker at p_5. Thus to stay sound we must consider that either t_1 or t_2 fired prior to t_3.

3.4 Relationships and Invariants over Values

Occasionally a protocol describes a message that is only allowed to be sent if for example some prior event has occurred or if some variable had a specific value in a prior or the current event's message. APSL was not expressive enough to allow capturing these dynamically dependent events. In order to support testing protocols that specify relationships between variables of different event instances, we had to extend our PN model with some notion of maintaining an environment during runtime and dynamically determining if an event is allowed to occur.

To enable this, we added an environment to each marker. A marker used to be defined with only a $p \in P$. It is now extended with an environment: p, Env. The environment can store mappings of variables to values. Transitions have been extended with a possible custom pre-condition and a possible sequence of statements: $(srcs, ev, dests, pre\text{-}cond, stmts)$. The $pre\text{-}cond$ and $stmts$ are allowed to remain undefined.

$pre\text{-}cond$ is a boolean expression and piggybacks on the expression grammar of AMSL. Variables of a marker's environment can be referred to from $pre\text{-}cond$. A transition can only fire when $pre\text{-}cond$ evaluates to $True$ for at least 1 marker per source place. If $pre\text{-}cond$ is undefined then it by default evaluates to $True$ regardless of the environments of the markers. Evaluating a $pre\text{-}cond$ over a marker's environment piggybacks on AMSL's expression evaluation functionality. A custom pre-condition could for example be defined as follows: $datalength <= 256 \mid\mid (datalength > 256 \;\&\&\; token == 5)$. The respective transition can then only fire if all of its source places have at least 1 marker the environment of which either contains a variable $datalength$ with a value that is at most 256, or contains a variable $datalength$ with a value that is at least 257 plus a variable $token$ with exactly value 5. If a $pre\text{-}cond$ refers to a variable that does not exist in a marker's environment, then the expression evaluates to $False$ for that marker.

If defined, the sequence of statements is evaluated upon firing a transition. This too piggybacks on AMSL by using its statement grammar and applying its evaluation functionality for statements. During evaluation, an environment is made available containing the respectively fired message's variables and their mappings to values. A sequence of statements could for example be: $datalength = msg.header.datalength;\ x = 4;\ y = x + 1$. Each statement is separated with a semicolon. In this example, the environments of the markers created at each destination place of the respective transition are modified with new value mappings for the variables $datalength$, x and y.

There are two unresolved issues. These are deficiencies that future work might want to address.

Firstly, with transitions that have multiple source locations it is not clear what the marker environments should be at the destination location(s) after firing the transition. We currently simply merge the environments and then pass the same environment to each destination marker. However, it might be required for certain protocols to specify per situation how the environments of each marker are passed on.

And secondly, the message generator currently cannot deal with specified relationships between variables and the environment. To enable this the generator needs to be extended with a constraint solving algorithm.

4 Petri Nets EDSL

Available Petri Net construction tools all opt for a WYSIWYG (what you see is what you get) approach. This is partly explicable by the graphical nature of Petri Nets. It is easy to visualize a PN, and when creating a PN one tends to recreate an already envisioned PN (or at the least step by step create partly envisioned PNs). These tools are all capable of exporting a specified PN to a file in a standard format. This makes it possible to specify a protocol using an already existing tool, exporting it to a file and letting APSL parse this file. However, specifying protocols directly with Petri Net transitions is a labor intensive approach which does not encourage good practices nor discourage bad practices. There are many repeating patterns present in protocol specifications that can be leveraged by a dedicated language. With this in mind we designed an EDSL for specifying protocols with the Petri Net data structure as a target language.

The EDSL is defined by some PN build instructions that internally call hidden functions to create new transitions and affect an internal build state $(PN, p_n, cursor, lbls, catching)$ in certain desired ways. PN contains the thus far built part of the Petri Net that is under construction ($PN : [srcs \xrightarrow{a} dests]$). Pointers are used to directly or indirectly refer to places of PN. A pointer can be defined in the following ways.

- An ID p. Thereby directly pointing to the respective PN place.
- A label that is mapped to a PN place in $lbls$ ($lbls : [Label \rightarrow p]$).
- The pointer p_n is a place holder for a new place. Dereferencing this pointer will return a new ID.
- The pointer $cursor$ is used to point to the current construction location and is modified by instructions to point to newly created (or occasionally already existing) places. An essential aspect of the EDSL is

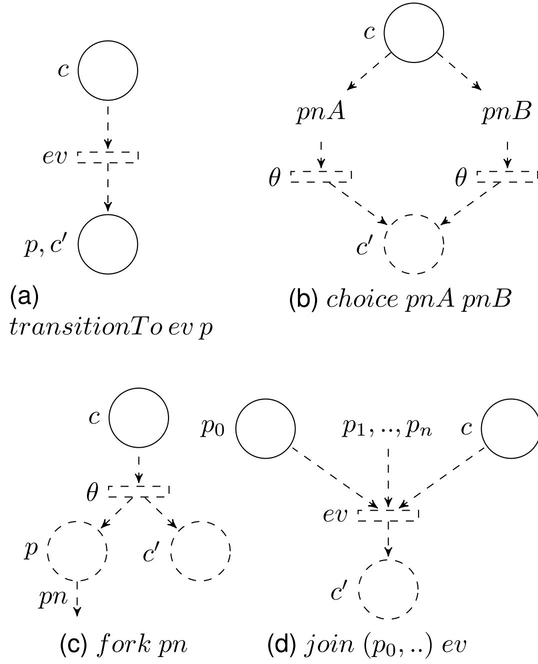


Fig. 16. A visualization of the effects of each basic instruction. $cursor$ is at c before evaluation and at c' after evaluation. Transitions, edges and places are either drawn solid when already existing prior evaluation or drawn dashed when created during evaluation.

this concept of a construction location. Instructions append transitions to this location.

It must be possible to define synchronization transitions (i.e. transitions with multiple incoming edges) that wait for markers from distinct locations. To enable this, pointers defined by unique labels point to the same place regardless of prior or subsequent instructions. Upon first time dereference of some label l (i.e. it is not yet linked to an ID p by $lbls$) it is linked to a new place (p_n). Concrete applications of this feature are discussed in a later part of this section.

The $catching$ variable in the build state stores a list of pointers. Upon every $cursor$ change, new transitions are added from the new $cursor$ towards each place that is pointed to in $catching$. This element of the build state is used by an instruction ($catch$, its exact functionality will be described in a later part of this section) that acts similar to the try catch functionality that many programming languages supply (e.g. c++).

The EDSL defines 4 basic PN build instructions. The essentials of each are summarized below. Fig. 16 visualizes the effects of the instructions.

- transitionTo :: Event → Pointer → Instructions
  $transitionTo\ ev\ p$:
  - Create a new transition: $[cursor] \xrightarrow{ev} [p]$
  - Update cursor: $cursor = p$
- choice :: Instructions → Instructions → Instructions
  $choice\ pnA\ pnB$:
  - Create a new place: $c' = p_n$
  - Store cursor: $origC = cursor$
  - Evaluate: $pnA;\ transitionTo\ \theta\ c'$
  - Restore cursor: $cursor = origC$
  - Evaluate: $pnB;\ transitionTo\ \theta\ c'$
  - The cursor must point to c' after evaluation of $choice$. This is already the case after the evaluation of the transitionTo instruction that is appended to pnB.
- fork :: [Instructions] → Instructions
  $fork\ pn$:
  - Create a new place: $p = p_n$
  - Create a new place: $c' = p_n$
  - Create a new transition: $[cursor] \xrightarrow{\theta} [p, c']$
  - Setup cursor for evaluation of pn: $cursor = p$
  - Evaluate: pn
  - Update cursor: $cursor = c'$
- join :: [Pointer] → Event → Instructions
  $join\ ps\ ev$:
  - Create a new place: $c' = p_n$
  - Add transition: $(cursor : ps) \xrightarrow{ev} [c']$
  - Update cursor: $cursor = c'$

In addition to $transitionTo$, a variant (transition :: Event → Instructions) is exposed that creates a transition to a new place. Furthermore, $continue$ (:: Event → Instructions) creates a transition from $cursor$ to $cursor$.

In addition to $choice$, for convenience the prelude library exposes a variant (choices :: [Instructions] → Instructions) that consecutively applies the choice instruction over a list of choices. Similarly, $forks$ consecutively applies $fork$ over a list.

Some miscellaneous functions are defined.

- $markInit$: adds the current $cursor$ to the list of start places of the Petri Net. This list determines the initial marking.
- $markExit$: adds the current $cursor$ to the list of exit places of the Petri Net.
- nop: no operation. The internal state remains unaffected by the evaluation of this instruction.

More complex instructions are defined in terms of the basic instructions. The EDSL currently defines five.

- loop :: Instructions → Instructions
  $loop\ pn$:
  - Determine the cursor before $loop$ evaluation: $origC = cursor$
  - Evaluate: $pn;\ transitionTo\ \theta\ origC$
  - The cursor must point to $origC$ after evaluation of $loop$. This is already the case after the evaluation of the appended transitionTo instruction.
- while :: Event → Instructions → Instructions
  $while\ ev\ pn$:
  - Evaluate: $loop\ (transition\ ev;\ pn)$
- parallel :: [Instructions] → Event → Instructions
  $parallel\ (seg_0 : segments)\ joinEv$:
  - Create a predetermined cursor place for each segment: $ps = \bigcup_{segments} p_n$

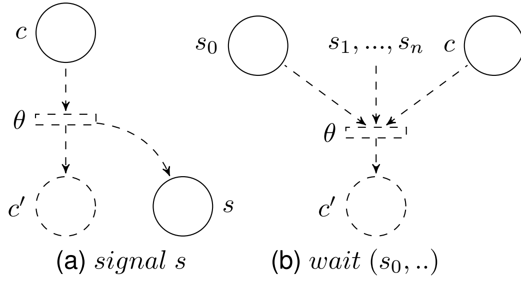


Fig. 17. A visualization of the effects of the two synchronization instructions. $cursor$ is at c before evaluation and at c' after evaluation. Transitions, edges and places are either drawn solid when already existing prior evaluation or drawn dashed when created during evaluation.

- Append each segment with a θ transition to their respective predetermined cursor: $segments' = \bigcup_{pn \in segments} (pn; transitionTo\ \theta\ p)$
- Evaluate: $forks\ segments'$
- Evaluate: seg_0
- Evaluate: $join\ ps\ joinEv$

- optional :: Instructions → Instructions
  $optional\ pn$:
  - Evaluate: $choice\ pn\ nop$
- catch :: Event → Instructions → Instructions
  $catch\ ev\ pn$:
  - Create a new place: $p = p_n$
  - Push p to the catching list: $catching = (p, ev) : catching$
  - Evaluate: pn
  - Pop p from the catching list: $catching = tail\ catching$

Additionally, two instructions have been defined that can be used to synchronize two or more in parallel running segments. For example, when a segment (seg_A) must wait until a segment (seg_B) finishes some sequence of actions, $wait\ [label]$ can be used in seg_A to wait until seg_B signals its completion of the respective actions with $signal\ label$. These patterns function exactly the same as countable semaphores do. Their implementations are briefly described below and Fig. 17 visualizes their effects.

- signal :: Pointer → Instructions
  $signal\ semaphore$:
  - Evaluate: $fork\ \theta\ [transitionTo\ \theta\ semaphore]$
- wait :: [Pointer] → Instructions
  $wait\ semaphores$:
  - Evaluate: $join\ semaphores\ \theta$

A definite advantage of the graphical Petri Net construction tools is that the PN is visually presented (even during construction). A visual feedback is a necessary tool to gain confidence in the PN model. Without, it is difficult to notice a mismatch between the model and the protocol's actual specification. And in the case of this EDSL, without any visual feedback it is easy to misunderstand the side effects of a particular instruction, further increasing the chances that the model is incorrect. Fortunately there exists a tool ($graphviz$) that can automatically visualize graph structures. Graphviz only requires the edges of the graph and then attempts to position each node optimally such that the edges intermingle minimally. Graphviz allows manually specified details (e.g. explicit positioning, labeling information or drawing style information) per node and edge, but it does not require this information. Using graphviz, a Petri Net can be automatically visualized by translating the net to a list of nodes that are either drawn with rectangles (transitions) or with circles (places). Additionally, a rectangle node is annotated with the event label of the corresponding transition. Fig. 19 presents a visualization of the PN that was constructed by evaluation of $examplePN$ (Fig. 18).

```
examplePN
  = do
  markInit
  transition "Login?"
  while "No!" $ do
    transition "Login?"
  transition "Logged in!"
  forks [control, pulseForth, pulseBack]
  transition "Logout?"
  transition "Logged out!"
  markExit

control
  = loop $ do
      transition "Request stuff?"
      transition "Received stuff!"

pulseForth
  = loop $ do
      transition "Server still alive?"
      transition "Server still alive!"

pulseBack
  = loop $ do
      transition "Client still alive!"
      transition "Client still alive?"
```

Fig. 18. An example application of the EDSL.

Unfortunately, Fig. 19 demonstrates that the extensive use of θ transitions by the instruction set of the language results in visualizations that are somewhat disconnected from the specifications that they represent. Most of these transitions can automatically be removed in an optimization process once the PN has been constructed, without affecting the semantics of the model (henceforth called θ-optimization). Consider for example the loop instruction. It appends a θ transition to a set of instructions to loop back to the original cursor location. This transition ($[p] \xrightarrow{\theta} [p']$) can be removed without affecting the semantics of the model by moving all incoming edges of p to p'. θ transitions that do not synchronize multiple markers (i.e. they have only a single incoming edge), and do not enable bypassing a semantical set of transitions, can be removed.

A place p can be removed if it falls within the following

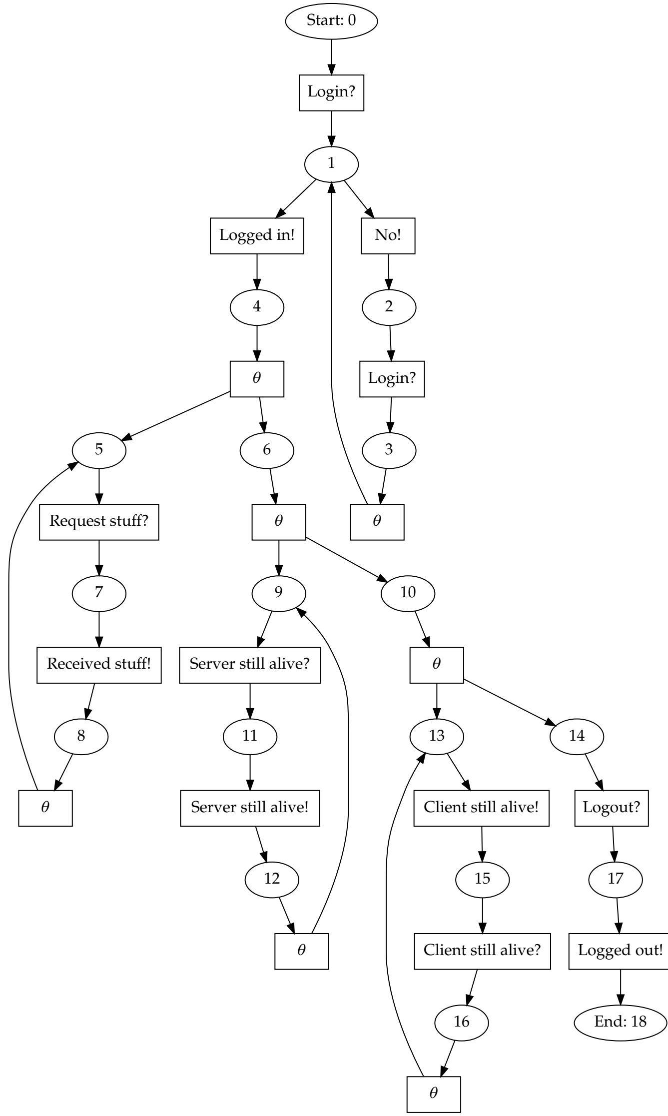


Fig. 19. The automatic visualization of an unoptimized PN using graphviz.

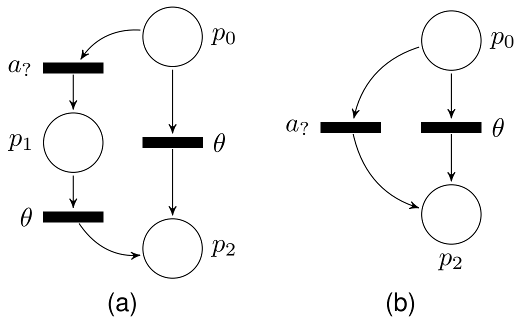


Fig. 20. A constructed PN prior and post optimization, visualized in (a) and (b) respectively. The θ transition from p_0 to p_2 in (a) cannot be removed without affecting the semantics of the model and remains unaffected.

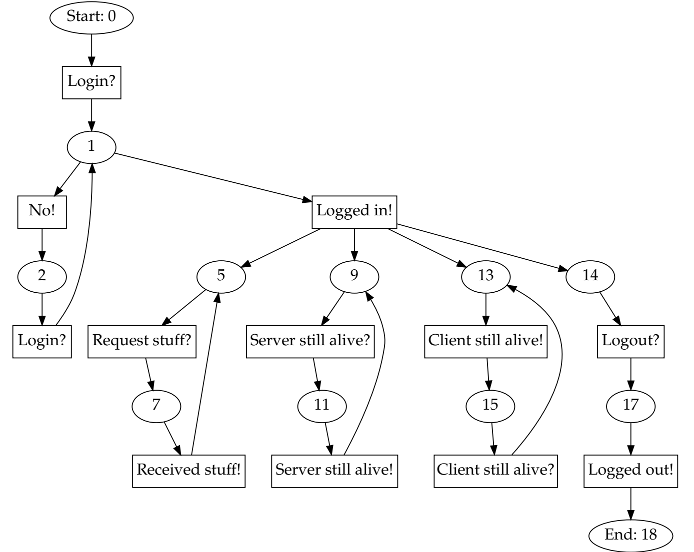


Fig. 21. The automatic visualization of an optimized PN using graphviz.

category.

$$p \notin PN.inits \ \wedge \ p \notin PN.exits \ \wedge$$
$$\forall t.\ t \in T \ \wedge \ p \in t.srcs : |t.srcs| = 1 \ \wedge \ t.ev = \theta$$

Upon removing a place p, the PN is modified as follows.

$$pDests = \bigcup_{t \in T \wedge p \in t.srcs} t.dests$$

$$\bigcup_{t \in T} \begin{cases} \emptyset, & p \in t.srcs \\ \bigcup_{p' \in pDests} \begin{array}{l} t_dests' = (t.dests \setminus [p]) \cup [p'] \\ t.srcs \xrightarrow{t.ev} t_dests' \end{array}, & p \in t.dests \\ \{t\}, & \text{otherwise} \end{cases}$$

Fig. 20 visualizes the effect of optimizing a θ transition away that was created during evaluation of $optional\ (transition\ a_?)$. This figure additionally presents a case where a transition cannot be removed. The θ transition from $[p_0]$ to $[p_2]$ makes the sending of $control_{a_?}$ an optional event. If this θ transition is removed $control_{a_?}$ is no longer an optional event, thus affecting the semantics of the model. Fig. 21 presents an optimized PN built from the same instructions as the unoptimized PN (Fig. 19) was built from.

It is still possible that some transitions can be removed at this stage. Some sequences of instructions result in θ transitions that loop from a place directly back to the same place. Consider for example the evaluation of: *loop nop*. This instruction will result in a θ transition from the head of the loop to the head of the loop. These instructions can be easily removed with a simple filter pass over the transitions of a PN. Transitions in the following category can be removed.

$$\forall t.\ t \in T \wedge t = [p] \xrightarrow{\theta} [p]$$

4.1 PNSL

The PN based APSL variant has been dubbed PNSL (Petri Net Specification Language). In order to use the EDSL for specifying protocols we need to enable specifying the events

event := message pre-cond? stmnts?

message := CName ’!’
message := CName ’?’
message := ”tau”
message := ”theta”

pre-cond := ’[’ Expr ’]’

stmnts := ’{’ stmnts_ ’}’

stmnts_ := stmnt
stmnts_ := stmnt ’;’ stmnts_

stmnt := LName ’=’ Expr

Fig. 22. The grammar of event strings. The grammars of CName, LName and Expr are defined by AMSL[1].

```
examplePN
  = do
  markInit
  transition "MessageA?"
  transition "MessageX! {x = msg.token}"
  choice (transition "MessageY!
                     [x == 1]
                     {c = x + msg.type}")
         (transition "MessageZ!
                     [x == 0]
                     {c = 2}")
  transition "MessageB? [c == 2 || c >= 5]"
  markExit
```

Fig. 23. The EDSL applied to specify a hypothetical protocol.

on each transition (that possibly contains a custom pre-condition and a sequence of statements).

The events are specified in string format in the PN build instructions. Once the PN has been constructed, each event string is parsed into our PNSL specific event data structure: $msg, pre\text{-}cond, stmnts$. The grammar of the event strings is shown in Fig. 22. Fig. 23 presents some examples of custom pre-conditions and statement definitions within the build instruction set.

5 Case Study: WebSocket Echo Server

5.1 The Specification

The code defining our specification ($Spec_{12}$) of the WebSocket protocol[10] is shown below. Fig. 24 shows a visualization of the specification.

```
1  wsPNBuilder = do
2    markInit
3    transition "ClientOpenHandshake?"
4    transition "ServerOpenHandshake!"
5    parallel [cli2serv,
6              serv2cli]
7             "theta"
8    markExit
9
10 cli2serv = do
11   parallel [cliPing,
12             cliData]
13            "MaskedClose? `cycle > 400`"
14
15 serv2cli = do
16   parallel [servPing,
17             servData]
18            "Close!"
19
20 cliData = do
21   loop $ do
22     catchTo "theta" (label "closeCliData") $ do
23       continue "MaskedMessageFrame?"
24       transition "MaskedMessageStartFrame?"
25       continue "MaskedContinuationFrame?"
26       transition "MaskedMessageEndFrame?"
27   transitionTo "theta" (label "closeCliData")
28
29 servData = do
30   loop $ do
31     catchTo "theta" (label "closeServData") $ do
32       continue "MessageFrame!"
33       transition "MessageStartFrame!"
34       continue "ContinuationFrame!"
35       transition "MessageEndFrame!"
36   transitionTo "theta" (label "closeServData")
37
38 cliPing = do
39   loop $ do
40     transition "MaskedPing?
41             `400 >= cycle`
42             {payload = msg.frame.payload_data}"
43     transition "Pong!
44             [payload == msg.frame.payload_data]"
45
46 servPing = do
47   loop $ do
48     transition "Ping!"
49     transition "MaskedPong?"
```

There are some interesting remarks regarding the specification code itself. We will first explain the structure of the WebSocket protocol and how its separate parts are defined by our specification and then explain some EDSL constructs that have not been introduced in Section 4 ($catchTo$ at lines 22 and 31, and the expression enclosed by backticks at line 13).

The WebSocket protocol defines an opening handshake and a closing handshake. The opening handshake is straightforward. The client sends $control_{ClientOpenHandshake?}$ and the server responds with $observe_{ServerOpenHandshake!}$ (this is described by lines 3 and 4). Upon finishing the opening handshake the client and server are both entering the exchange of data section of the protocol. In this section, both the client and the server can send data to each other concurrently.

$cliData$ (lines 20 to 27) describes the sending of data from the client to the server. The protocol specifies that large data can be divided into multiple smaller packets. A multi-packet message must be started with a StartFrame and ended with an EndFrame (and with zero or more

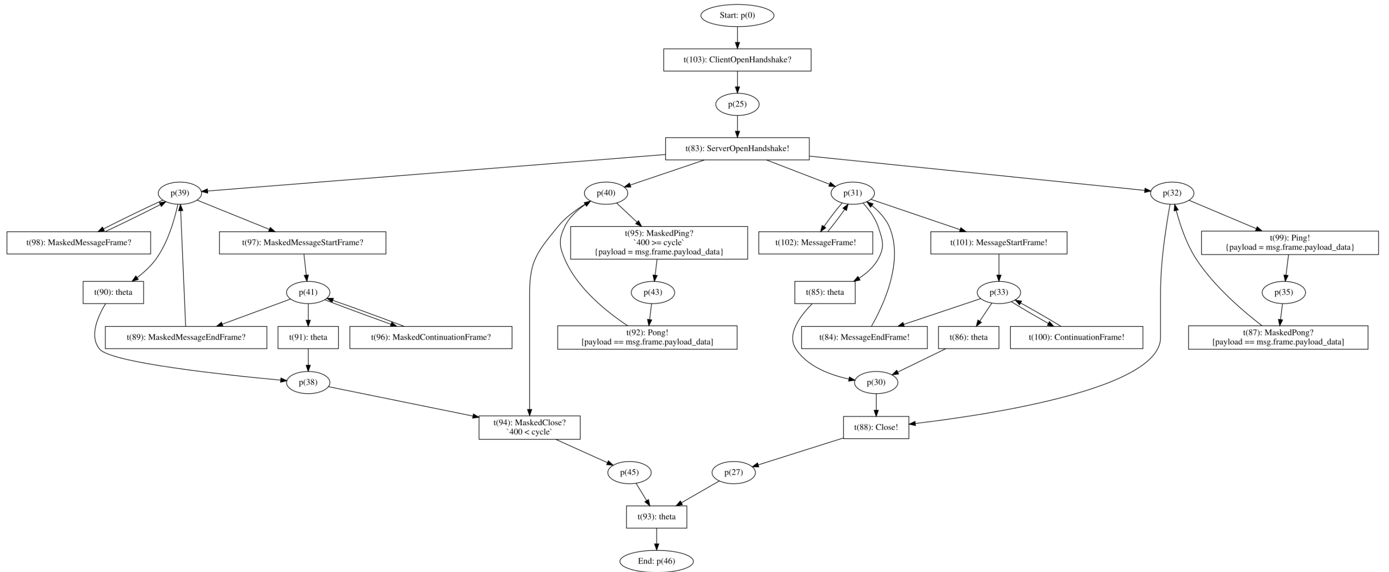


Fig. 24. A visualization of $Spec_{12}$, specifying the WebSocket protocol. A higher resolution render of the same visualization is shown in Appendix A.

ContinuationFrames in between). $servData$ (lines 29 to 36) similarly defines the server's side of sending data. The protocol additionally defines that the data inside messages (i.e. everything except the message header) that are sent by the client must be masked (hence the Masked_ frames in $cliData$, and the regular frames in $servData$).

When we are in the data transmission section of the protocol the participants are allowed to concurrently ping each other (e.g. to check if the other party is still reachable). $cliPing$ (lines 38 to 44) describes the client side initiated ping. The protocol defines that a ping message may contain data. The response pong message must contain the same data (this is a sanity check, if a participant fails to send a pong conforming to this requirement the other may conclude that it should close the communication immediately). $servPing$ (lines 46 to 49) similarly describes the server side initiated ping. Unfortunately, we haven't added functionality to construct messages that conform to some custom pre-condition expression. However, the pong message that is sent by the client (line 49) should contain the same data as the ping message that priorly was sent by the SUT.

Finally, the closing of the connection is a bit peculiar. Essentially, a connection is closed after the client has sent $control_{MaskedClose}$ and the server has sent $observe_{Close}$. However, it does not matter who sends this close message first. Furthermore, after a party has sent his close message, it may no longer send any data frames or pings. According the protocol, the other party does not immediately have to respond with a confirming close message. It may continue sending data frames and pings for as long as it wants. Although it should not expect that the other side is still processing these messages. And last, a party may send its close message at any time after the opening handshake has been finished. Thus, a close message may be sent before finishing a multi packet data message. Our specification describes this closing handshake mechanism by removing the markers that are essentially in $cliData$ and $cliPing$ when the client sends $control_{MaskedClose}$ (this is described in lines 10 to 13). By merging these markers at this event, we disallow the client to send more data frames and pings after $control_{MaskedClose}$ is sent. Similarly, the markers that are essentially in $servData$ and $servPing$ are removed upon receiving $observe_{Close}$ (lines 15 to 18). Once both close messages have been registered, there will be a marker at p_{45} and a marker at p_{27}. From this configuration we may reach the end location of the protocol by firing the θ transition to p_{46}.

The event string at line 13 of the specification has a construct that has not been formally introduced in Section 4. The expression enclosed by backticks is essentially a guard of the transition, it differs from the pre-conditions that are enclosed by brackets (e.g. line 44). Pre-conditioned transitions can be expected to fire whenever the transition's source places contain a marker. Upon registering the event, the pre-condition is evaluated which determines if the message content is correct (corresponding to some prior communicated information). The guards determine if a transition is enabled. In $Spec_{12}$ the transition on $control_{MaskedClose}$ is enabled after the 400th communication step. We added this prerequisite to the transition to prevent the tester from shutting the communication down (almost) immediately after opening the connection.

Furthermore, the $catchTo$ instruction has not been defined in Section 4. As it stands, it has been implemented as a hack. It essentially acts like the $catch$ instruction does, but instead adds catch transitions that transition towards a manually specified place, and (here comes the catch) it leaves the tail pointer unaffected. The desired way of defining cliData would be as follows.

```
cliData = do
  catch "theta" $ do
    loop $ do
      continue "MaskedMessageFrame?"
      transition "MaskedMessageStartFrame?"
```

```
continue "MaskedContinuationFrame?"
transition "MaskedMessageEndFrame?"
```

And the same (i.e. applying $catch$, instead of $catchTo$ in conjunction with an extra $transitionTo$ instruction) applies to the definition of servData. However, if we specify these segments this way, we unfortunately get a PN (dubbed $Spec_{12}'$) that has two additional transitions: $[p_{41}] \xrightarrow[\text{Spec}]{MaskedMessageEndFrame_?} [p_{38}]$ and $[p_{33}] \xrightarrow[\text{Spec}]{MessageEndFrame!} [p_{30}]$ (both arise during θ-optimization). It is essential to notice that these additional transitions do not change the semantics of the specification. With both specifications $Spec_{12}$ and $Spec_{12}'$ sending the end frame at p_{41} may result in a marker at p_{39} or a marker at p_{38}. However, under the former the marker can only reach p_{38} through the firing of θ transitions. Under the latter specification, the marker can reach p_{38} without firing θ transitions. This essentially undermines the lazy θ firing strategy. For example, assuming we are using $Spec_{12}'$ and we have $\mathbf{C}\{[p_{41},..]\}$ after some initial communication trace, if the tester sends $control_{MaskedMessageEndFrame}$ we get $\mathbf{C}\{[p_{39},..],[p_{38},..]\}$. Whereas if we are using $Spec_{12}$ we get $\mathbf{C}\{[p_{39},..]\}$. The extra transitions in $Spec_{12}'$ can probably be removed during θ-optimization. However, it is not entirely certain how to automatically find and remove these transitions without affecting the semantics of the Petri Net. Improving the automatic optimization of Petri Nets might be an interesting topic for future work.

Examination of the formal specification document of the WebSocket protocol will show that our specification code in the EDSL language is quite similar in structure. This indicates that we do not need to introduce artificial control behavior into the specification code in order to generate a Petri Net that represents the formal document correctly. Subsequently, this reduces the odds that errors are introduced during the modeling process that would result in mismatches between the Petri Net specification and the protocol's formal specification document.

5.2 Test Results

After running 5 test iterations we reached a transition coverage of 85.71% (18 out of 21 transitions of the specification fired at least once). It took about 12 minutes to perform these 5 repetitions. Table 11 shows how many times each transition was fired. Two of the three transitions that never fired (transitions 87 and 99) did not fire because the SUT never sent a ping message to the tester. This has purposefully been disabled in the SUT, since APSL needs a constraint solver before it is able to construct a correct MaskedPong message (that contains the same payload as the ping message that it responds to). The third transition that never fired (transition 86) did not fire because apparently after the tester has sent the $control_{MaskedClose}$ message, this particular SUT always first finishes transmissioning a multi packet message before signaling the closing of its side of the communication with a responding $observe_{Close}$ message.

Had these tests been run with a test tool that can only soundly track *synchronous* communication, we would have reached a significantly smaller number of actions performed within the same time frame. In this setup, to gain some certainty that messages are not sent simultaneously by both parties, we have to introduce a pause wherein we wait for any sent messages from the SUT to arrive (e.g. 500ms, or 250ms, or some other number that should be as low as possible while maintaining certainty that asynchronous effects will not be encountered). And, arguably much more importantly, we would have no soundness guarantee of the state tracking process, thus any error messages that the test tool would produce are possibly caused by asynchronous effects and consequently do not show a mismatch between the specification and the implementation. This might subsequently cause a lowered trust in the overall test results that the tool produces by the engineers who apply APSL.

Had these tests been run with a test tool that considers asynchronous effects, but uses an LTS based specification, we would not have been able to ever apply the optimization technique after sending control messages. WebSocket has one concurrent segment wherein zero observe messages are sent. As a result, the tester never reaches a point where it can conclude that the sent control messages in that concurrent segment have certainly arrived (at least, not until it sends a MaskedClose message and receives the confirming Close message). This causes a state explosion. For example, after having sent 10 control messages from the MaskedFrame concurrent segment, the state tracker would have built up an $\mathbf{S}\{..\}$ containing 11 possible states. The first possible state considers that all 10 control messages have not arrived yet, the second possible state considers that all but the first control message have not arrived yet, the third possible state considers that all but the first 2 control messages have not arrived yet, ect. This not only makes it more difficult for the test strategy to effectively make sound decisions, it also quickly explodes $\mathbf{S}\{..\}$. A laptop containing 4GB of RAM could only effectively reach performing 30 actions, after which tracking $\mathbf{S}\{..\}$ began requiring too much RAM.

The difference between the actual performed test that applies a PN based specification with these hypothetical tests is significant. The PN based specification exhibits no

Transition ID	Transition event	Fire count
83	$ServerOpenHandshake_!$	5
84	$MessageEndFrame_!$	324
85	θ	5
86	θ	0
87	$MaskedPong_?$	0
88	$Close_!$	5
89	$MaskedMessageEndFrame_?$	248
90	θ	3
91	θ	2
92	$Pong_!$	86
93	θ	5
94	$MaskedClose_?$	5
95	$MaskedPing_?$	86
96	$MaskedContinuationFrame_?$	236
97	$MaskedMessageStartFrame_?$	250
98	$MaskedMessageFrame_?$	271
99	$Ping_!$	0
100	$ContinuationFrame_!$	399
101	$MessageStartFrame_!$	324
102	$MessageFrame_!$	195
103	$ClientOpenHandshake_?$	5
Sum count		2454

Table 11
The number of times each transition of $Spec_{12}$ fired during a test run consisting of 5 test repetitions.

non-deterministic behavior and no implicit asynchronous effects (i.e. all asynchronous effects are modeled explicitly resulting in the four parallel segments). As a result, APSL was able to apply the optimization technique after every sent control message. Subsequently, $\mathbf{S}\{..\}$ never consisted of more than 1 possible state.

There are two additional benefits that have not yet been discussed that arise from using the PN based formalism. Consider for example $Spec_{12}$ and assume that the opening handshake has been performed (the communication trace thus far is $ClientOpenHandshake_? \cdot ServerOpenHandshake_!$), and the communication state is: $\mathbf{C}\{[p_{39}, p_{40}, p_{31}, p_{32}]\}$. There is only 1 possible configuration, and it contains multiple markers that can advance on a control message (the marker at p_{39}, and the marker at p_{40}).

The first advantage is that when there are multiple markers that can advance on control messages, we can advance them all in a single communication step of the tester (i.e. we do not have to act on incoming observe messages in between sending each of these control messages). Thus, the tester can actually take advantage of the protocol's concurrent behavior, and establish a higher bandwidth.

The second advantage is that when there are concurrent segments and when we are using the PN based formalism, we can aim for a cleaner transition coverage compared to when we are using the LTS based formalism. It can be (and probably is) extremely difficult to trigger all state transitions in an LTS that implicitly models concurrent behavior. This is because we are then trying to reach all states that implicitly represent some unique combination of the concurrent segment's states. Which combination (and thereby which state of the LTS) is reached heavily relies on aspects that are not captured by the LTS model (e.g. timing). For example, consider again the basic example specification $Spec_6$ of Section 2 with its two implicitly represented concurrent segments $Spec_{6A}$ and $Spec_{6B}$. If we want to reach full transition coverage, we have to first be able to reach all states. Reaching state s_3 means that the tester has to immediately send $control_b$ after having sent $control_a$, at least before receiving $observe_x$. whether this is achievable depends on the transition speed in the test environment, on how fast the SUT processes and acts on incoming control messages, and on how fast the tester can generate and send $control_b$. Compare this to MBT using a PN based formalism and assume again the similar specification $Spec_9$ of Section 3. The concurrent segments are now modeled explicitly. In order to reach full transition coverage, we have to first be able to reach all *places*. However, we do not have to be able to reach all possible states (where the term state here is equal to the term configuration) in order to reach full transition coverage. Reaching the places of the protocol with markers is something that usually does not require information other than the Petri Net specification.

We can still try to reach full state coverage by reaching all possible combinations of marker positions, although this is a goal that is probably not feasible for the average asynchronous communication protocol. It is key that by applying MBT based on a PN formalism, we do not require achieving this infeasible target in order to obtain a high transition coverage.

6 Related Work

6.1 Automatic Test Tools

TGV [2] automatically generates test suites (i.e. a set of test cases) for testing conformance between the specification and implementation of synchronously communicated protocols. The tool's inputs are an LTS (describing the formal specification of the protocol), and a test purpose. A generated test suite consists of a set of test cases, each describing step-wise what a tester must do and expect from the SUT based on what has previously been observed.

TorX [4] similarly automatically generates test suites based on a formally specified protocol (in a format similar to LTS), and (optionally) a specified test purpose. It subsequently automatically performs the tests and verifies the results with the expected results. Similar to our approach with APSL, both TorX as well as TGV compute the allowed continuation actions and expected SUT behavior online (i.e. while testing) based on the current set of possible states. TorX can either follow a test strategy based on a manually specified test purpose (these are similar to TGV's test purposes) or based on a generic test purpose that has been derived from a set of heuristics to maximize fault finding probability.

Our work has established a tool that is in many ways similar to TGV and TorX. TGV and TorX both apply more advanced and manually specifiable test strategies than APSL currently does. However, we have enabled the testing of asynchronously communicated protocol implementations which is something that neither TGV nor TorX can do in a sound approach.

6.2 Formal Specification of Asynchronous Communication Protocols

There has been some research performed that focused on enabling automatic testing of asynchronous protocols or finding formal models to specify these protocols in a natural way.

[6] uses a set semantics (captured by a Prime Event Structure) to model equivalence between the specification and observed SUT behavior. The set semantics cannot discriminate between the order of arrived messages within channels, it can only relate observations with the specification by verifying that all prerequisite events (required for this observation to be an allowed event) have been met. This allows their work to be applied to asynchronous communications that deal with imperfect channels (i.e. channels that cannot guarantee that sent messages are received in the same order by the receiving participant). We were inspired for our work by Tretmans' queue semantics which assumes perfect communication channels, but by doing so we were able to find an essential optimization of the state tracking process.

[11] defines an algorithm to transform a set of LTS models (one for each communication participant) into a Prime Event Structure, and has been referred to by [6] as a possible technique to obtain the set semantics. This technique is capable of capturing the interleaving of messages sent over different channels. However, it is incapable of capturing explicit information about concurrent segments into the Prime Event Structure.

Recent work [12] has designed a protocol specification language (dubbed *Scribble*) that in many ways is comparable to our EDSL. With Scribble the authors have strived to enable specifying communication protocols (that may include concurrent segments) in a natural way. Scribble does not separate the specification of message types (our AMSL) from the specification of state transitioning (our PNSL), both are specified within the same construct. Scribble compiles towards a session type structure, which builds its formal basis on the π-calculus and the type theories that have arisen from the π-calculus. It is unclear if our formalism that we based on Petri Nets is weaker or stronger compared to their π-calculus based formalism. However, at the least by basing Scribble on session types they can statically determine type correctness of a protocol specification, which neither AMSL nor PNSL can currently do.

7 Future Work

Enabling automatic testing of asynchronously communicating protocol implementations has opened many doors for future work. It is especially interesting to further research applicability of the APSL tool by defining test strategies. Section 7.1 defines two different kinds of test strategies that could be considered for future work.

Furthermore, it might be interesting to find other formal paradigms that can serve as a viable underlying basis for MBT of asynchronously communicated protocol implementations, and to compare these with our current Petri Net basis.

And finally, future work may address some engineering deficiencies of our tool:

- The message generator should be equipped with a constraint solver to enable generating message instances that are according to specified relationships between its variables and markers' environments.
- PNSL can be improved to enable specifying custom marker environment passthrough methods. This is probably a desired capability when specifying transitions that merge multiple markers.

7.1 Test Strategy

There are two distinct reasons why we might test a system. The first is to find faults, and the latter is to establish general confidence that a recent version of the system is reliable enough. This section first explains that these goals are fundamentally different and should not be performed by the same set of tests. In short, as Hamlet has stated:

> Tests designed to expose failures are not representative, and tests representative of typical operation may seldom encounter failure situations. [13]

7.1.1 Testing for Operational Reliability

The MTBF (Mean Time Between Failure) of software can be estimated in a similar way the MTBF might be estimated for hardware systems. By testing software with realistic inputs iteratively many times, without encountering any failures, the confidence in the reliability of the software (or any type of system) objectively increases. Intuitively this may seem incorrect because software by nature acts differently compared to hardware. Hardware systems (not specifically limited to hardware that runs software) consist of many smaller parts composed together, each component can wear and tear over time and each component (directly) only affects a small amount of neighboring components. Software does not wear or tear over time (we are neglecting the astronomically low chance that a bit might flip), and each component can affect many other components that can be located anywhere within the system. However, Parnas et. al state why estimating MTBF of software is meaningful:

> When a program first fails to function properly, it is because of an input sequence that had not occurred before. The reason why software appears to exhibit random behavior, and the reason that it is useful to talk about MTBF of software, is because the input sequences are unpredictable. When we talk about the failure rate of a software product, we are predicting the probability of encountering an input sequence that will cause the product to fail. [14]

Hamlet explains in [15] how we can compute an estimation of a software's reliability. If we run n tests with randomly picked inputs, and do not encounter any failures, we establish a confidence that the next m number of randomly picked inputs will not encounter failure with an $i\%$ likelihood (the exact relation between n, m and i is explained in [15]). We emphasize again that it is important that the inputs for each test are randomly picked from a distribution that reflects realistic inputs the SUT encounters under actual operation. Otherwise, we are acquiring confidence in the SUT while it is running in an environment that does not represent the environment wherein it will be deployed.

A large amount of work in testing theory has focused on reaching some coverage measure with a least number of test cases. This is done by systematically targeting a distinguished part of the system with each test case (also called partitioned testing of the input space). Striving for reaching similar test quality with a smaller amount of test cases has been a response to scaling issues and a lack of control when otherwise randomly testing a system. Covering all or a significant part of the areas of a system's capabilities during random testing requires an excessive amount of test cases for anything other than the simplest of systems. While these partition testing techniques can reach higher fault detections with fewer test iterations compared to random testing, they cannot contribute to truly estimating the reliability of the system under operation in the field. Hamlet has stated that this is a fundamental limit of testing aimed at finding faults:

> The ability to quantify a test that does *not* fail is unique to random testing. [15]

Tests that were designed systematically to target specific areas of the input cannot contribute to estimating reliability because these tests do not reflect a realistic application of the system.

> Tests seeking to excite failure are not representative of a program's day-to-day operation. Confidence in daily performance can be gained only by testing that mimics the "operational distribution" of typical usage. [13]

Arguably, we must refrain from obtaining a sense of confidence in software from successful results of tests that were

applied following a fault seeking technique.

Unfortunately, in recent years (i.e. about the last two decades) little attention has been paid to this. This can partly be explained by the conclusions that these authors eventually reached concerning the requirements put to testing for true confidence.

The first conclusion concerns the excessive amounts of test iterations that are required to establish an acceptable estimation of reliability. This was simply not practically achievable with the hardware and software that were available at the time. However, with APSL we can fully automatically test an implementation once the protocol has been specified. This means that we can easily scale the test environment to multiple testers that are communicating with 1 or multiple SUTs simultaneously, thereby reaching high numbers of test iterations per hour. Assuming that the following two concerns can be addressed, a setup like this might enable reaching meaningful estimations of reliability.

The second conclusion concerns the need for an operational profile. This profile describes the distribution of inputs that the SUT will encounter when it is running in the field. Some research has been performed that looked into how the operational profile can be acquired. [16] presents a process where the operational profile is incrementally built and modified based on expert domain knowledge. Acquiring the operational profile through this method unfortunately requires a large amount of manual work, but it has been shown to produce accurate profiles of the system's behavior. There might be an opportunity here for APSL to automatically generate an operational profile, by observing conversations between systems that are already running in the field. However, there seems to be little related work that this hypothetical method could expand on or learn from. It is currently very difficult to estimate whether this is practically achievable.

And lastly, the third conclusion concerns the state based nature of communication protocols. Essentially, the results of sending a control action to the SUT probably affect and are probably affected by a global state of the SUT. And this state might not be reset for days or even years. If we want to estimate the reliability of the SUT in its entirety we have to consider the sequences of messages that are likely to occur when constructing the operational profile, which is an aspect that seems to explode the number of parameters of the operational profile. It might be possible to circumvent this issue by instead testing for reliability of each transition (thereby estimating the MTBF of firing a transition). It might then be possible to derive a global MTBF estimation of the implementation from the determined MTBFs of the transitions. Although, according to Parnas et al. this is a temptation that should be avoided:

> The essence of system-reliability studies is the computation of the reliability of the parts. It is tempting to try to do the same thing for software, but the temptation should be resisted. ... the effect of one execution [of some part of the system (e.g. firing a transition)] depends on the state that results from earlier executions. One failure at one part of a program may lead to many problems elsewhere. [14]

However, I do not necessarily agree with this. This seems to only increase the unpredictability of the inputs that can arise (an input is then a tuple containing the global state of the SUT plus the communication message), thereby only further increasing the stochastic behavior that the software seems to exhibit.

The work of Hamlet et al. in [17] can give initial pointers about how we might compute a global estimation of reliability that is based on established reliabilities of each component (i.e. each transition) of the software system. An essential aspect of his proposed method is that it takes in consideration the structure of how the components are composed.

7.1.2 Testing for Failures

There are good reasons why partition testing is a useful and necessary technique.

> Catastrophic failures deserve an extra prevention effort, and partition testing is the only way to attack them. These failures may be so infrequent that they are discovered in use only when it is too late, but a partition can be constructed to seek them. [13]

Ntafos has expanded on this.

> Very low probability subdomains are likely to be disregarded by random testing. While that may not affect the operational reliability much, it could have a large impact if the cost of failures in such low probability subdomains is very high. This is a major flaw of random testing and a good reason for using it as a complementary rather than the main/only testing strategy. [18]

Partitioning the input space optimally such that the most errors of an implementation are located in the shortest amount of time is a complex search problem. A lot of research has been performed in various areas that can be expanded on or learned from to find viable fault seeking strategies for APSL.

It is essential to notice that with APSL we can target much more detailed coverage metrics than has previously been possible. The message descriptions of each message type that AMSL supplies, enable tracking what value ranges have or have not yet been covered for each variable. This potentially enables significant improvements in existing fault seeking strategies.

Each fault that is present in a protocol's implementation can essentially be discovered by setting a range of "interrelational parameters"[19] to certain values. Here, the interrelation keyword describes that in some way their combined configuration affects the implementation's response(s). Thus, if any of those parameters would be set to some other value the respective invalid behavior would not occur. Usually, the "maximum size" of interrelational parameters is much lower than the system's total amount of parameters. Combinatorial Testing (CT) finds significant performance improvements by assuming that all parameters are at most affected by n other parameters. In CT, the tests are generated from a *covering array*. Each row of the covering array represents a test case. The columns specify the value for each parameter, respectively. There are several

ways to generate the covering array, each having different characteristics in what kind of combinations can be found at what test suite execution time cost. [19] describes several of these strategies.

EvoSuite[20] is a good example of how a well constructed Genetic Algorithm can find the smallest possible test suite for a Java class that still satisfies a chosen coverage criteria. The Genetic Algorithm incrementally approaches its goal by each time mutating and crossing over test suites of the current generation (i.e. the current iteration).

Fittest[21] shows how complex search problems can be addressed with a combined approach of multiple techniques. The tool can automatically perform regression tests on Future Internet applications. These applications often present sophisticated user interfaces that are constantly modified to improve the user's experience. Keeping a specification manually up to date along with these modifications is an error prone and time consuming task. Fittest can automatically infer the changes to the specification after each new application version and subsequently automatically apply its regression tests. However, in order to accomplish this it has to find solutions in large search problems. Although protocol implementations are different by nature compared to Future Internet applications (or to user interfaces in general), APSL is faced with similar complex search problems. The strategy that is applied by Fittest is likely relevant to viable strategies that can be applied by APSL. Fittest combines a number of relevant techniques, including Combinatorial Testing and Genetic Algorithms. When applying a Combinatiorial Testing strategy, it is difficult to find an efficient coverage array. Fittest finds this by first generating a Java program that represents the problem and subsequently applying EvoSuite to find a suitable solution.

The thus far mentioned techniques are positive testing techniques. They generate test cases that communicate validly according to the specification. The other kind of partition testing techniques generate negative test cases. These test cases purposefully are not valid according to the specification, and are probing the SUT to find holes in its security. It can be especially interesting to purposefully violate the constraints that are placed between variables of messages. For example, the Heartbleed[22] bug that was present in a widely applied SSL library presents an interesting case where violating a constraint placed between two variables in the ping message would result in information being returned by the server in the pong message that was supposed to remain secret. With negative tests that target these constraints, APSL can automatically detect the Heartbleed bug (and other comparable bugs): if the SUT responds as if the constraint was not violated, APSL can conclude that a flaw in the implementation has been found.

Recent work [23] has shown that with a tool like APSL, it is possible to collect detailed information about the thus far communicated messages during the application of a test case. With the proposed approach it is feasible to then efficiently query sophisticated properties on the collected information. This enables complex test strategies to be defined that base desisions (which are made during a test) on queried information. For example, it would be possible to specify a test strategy that guarantees that instances of control messages are never repeated during each single test case. Or that a certain message field's value (e.g. the value of field x in message $msgA$: $msgA.x$) should be assigned unique values during a test case. Or conversely that there should be a point in the communication trace where a certain message instance's field is assigned a value that was already used in a prior message. Although these examples are still quite straight forward, it becomes more complicated if, for example, we want to query this information in conjunction with what places have (or perhaps have not) been visited by markers. It would be interesting to, in the context of PNSL, research applicability of (complex) queries on information gathered using the approach of [23] to find succesful failure seeking test strategies.

It is possible that a long test sequence ends in a failure (i.e. the SUT sent a message that it was not allowed to send). It is then useful to first find a smaller test sequence that still reproduces this failure before analyzing what is causing the failure. In [24] the authors discuss a debugging technique that, after a failure is found, can automatically find shorter sequences that still reproduce the failure. Although their work assumes that the SUT is deterministic, it would be interesting to research how well it can perform in the context of PNSL (which might encounter highly non-deterministic SUTs).

8 Conclusion

An essential optimization was found concerning the automatic tracking of the communication state under asynchronous communication. We have shown that without this optimization, tracking the state of asynchronously communicated protocols is infeasible. In order to reach maximum application of the optimization, we propose modeling the specification using a formal model that can explicitly represent in parallel traversed segments.

Our proposed solution involved using a Petri Net as a formal model of the protocol specification. However, the established results are probably achievable with any formal model that enables an explicit representation of in parallel communicated segments. Further research may find representations that better fit the domain of MBT.

An EDSL was designed that translates a combination of build instructions to a Petri Net data structure. The tester can subsequently base its testing on the generated Petri Net. The EDSL has enabled specifying protocols in a natural way, including cases in which the protocol contains many in parallel communicated segments and synchronizations between these segments. Subsequently, protocol specifications in the EDSL language can be similar in structure as the official protocol's description document. This reduces the necessary engineering effort to specify the protocol, and thereby also reduces the proneness to errors of the specification process.

Importantly, we have shown that automatic model based testing of asynchronously communicated protocols is achievable.

References

[1] T. Tervoort and I. S. W. B. Prasetya, "APSL: A light weight testing tool for protocols with complex messages," in *13th International Haifa Verification Conference (HVC)*. Springer International, 2017, pp. 241–244.

[2] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 297–315, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10009-004-0153-x

[3] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites," in *International Conference on Computer Aided Verification*. Springer, 1996, pp. 348–359.

[4] J. Tretmans and E. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43. [Online]. Available: http://doc.utwente.nl/66990/

[5] G. J. Tretmans, "A formal approach to conformance testing," Ph.D. dissertation, Twente University Press, 1992.

[6] P. P. Salas and P. Krishnan, "Automated software testing of asynchronous systems," *Electronic notes in theoretical computer science*, vol. 253, no. 2, pp. 3–19, 2009.

[7] H. Zhu and X. He, "A methodology of testing high-level petri nets," *Information and Software Technology*, vol. 44, no. 8, pp. 473–489, 2002.

[8] C. A. Petri, "Kommunikation mit automaten," 1962.

[9] E. W. Mayr, "An algorithm for the general petri net reachability problem," *SIAM Journal on computing*, vol. 13, no. 3, pp. 441–460, 1984.

[10] I. Fette, "The websocket protocol," 2011.

[11] O. Henniger, "On test case generation from asynchronously communicating state machines," in *Testing of Communicating Systems*. Springer, 1997, pp. 255–271.

[12] K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Deniélou, and N. Yoshida, "Structuring communication with session types," in *Concurrent Objects and Beyond*. Springer, 2014, pp. 105–127.

[13] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence (program testing)," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, 1990.

[14] D. L. Parnas, A. J. Van Schouwen, and S. P. Kwan, "Evaluation of safety-critical software," *Communications of the ACM*, vol. 33, no. 6, pp. 636–648, 1990.

[15] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 1994.

[16] J. D. Musa, "The operational profile," in *Reliability and Maintenance of Complex Systems*. Springer, 1996, pp. 333–344.

[17] D. Hamlet, D. Mason, and D. Woit, "Theory of software reliability based on components," in *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 2001, pp. 361–370.

[18] S. C. Ntafos, "On comparisons of random, partition, and proportional partition testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 949–960, 2001.

[19] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[20] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Quality Software (QSIC), 2011 11th International Conference on*. IEEE, 2011, pp. 31–40.

[21] T. Vos, P. Tonella, W. Prasetya, P. M. Kruse, A. Bagnato, M. Harman, and O. Shehory, "Fittest: A new continuous and automated testing process for future internet applications," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 407–410.

[22] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE security & privacy*, vol. 12, no. 4, pp. 63–67, 2014.

[23] I. Prasetya, "Temporal algebraic query of test sequences," *Journal of Systems and Software*, vol. 136, pp. 223 – 236, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121730153X

[24] A. Elyasov, W. Prasetya, J. Hage, and A. Nikas, "Reduce first, debug later," in *Proceedings of the 9th International Workshop on Automation of Software Test*, ser. AST 2014. New York, NY, USA: ACM, 2014, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/2593501.2593510

Appendix A