An Approach to T-way Test Sequence Generation With Constraints

Feng Duan, Yu Lei
University of Texas at Arlington
Arlington, TX 76019, USA
feng.duan@mavs.uta.edu,
ylei@cse.uta.edu

Raghu N. Kacker, D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn}@nist.gov

Abstract—**In this paper we address the problem of constraint handling in t-way test sequence generation. We develop a notation for specifying sequencing constraints and present a t-way test sequence generation that handles the constraints specified in this notation. We report a case study in which we applied our notation and test generation algorithm to a real-life communication protocol. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for the protocol. However, the test generation algorithm takes a significant amount of time. This work is part of our larger effort to make t-way sequence testing practically useful.**

Keywords-Test Sequence Generation; Sequencing Constraint; T-way Sequence Coverage; Sequence Testing; Event-based Testing; Combinatorial Testing

I. Introduction

Many systems, e.g., interactive systems [1], event-driven systems [2] and communication protocols [3], exhibit sequencing behavior, where a sequence of events is exercised during each execution and the order in which the events occur could significantly affect the system behavior. To test these systems, we need to generate test sequences, in addition to test data.

T-way sequence testing applies the notion of *t*-way coverage to test sequence generation [4]. Informally, given any *t* events, if they could be exercised in a given order, there must exist at least one test sequence in which these events are exercised in this order. Doing so allows us to test all possible interactions between any *t* events. Thus, *t*-way sequence testing can expose faults that are caused by interactions between no more than *t* events.

One important problem in *t*-way sequence testing is dealing with sequencing constraints [5], i.e., restrictions on the order of events that must be satisfied for a test sequence to be valid. The technical challenge is two-fold. First, a notation is needed to specify sequencing constraints. This notation must be easy to use and have the power to express commonly encountered constraints. Second, a test generation algorithm must be developed to handle sequencing constraints. Compared to constraints on data values, sequencing constraints can be more difficult to handle. This is because the space that needs to be searched in the evaluation process for sequencing constraints can be much larger due to the extra dimension, i.e., order of events.

Recent years have seen significant progress on *t*-way test data generation, but not on *t*-way test sequence generation [6]. Kuhn et al. [4] presented an approach to generate SCAs (Sequence Covering Arrays) for testing special systems. Their approach requires each event occurs exactly once in each test sequence, and only supports one type of constraint. Yu et al. [7][8] presented another approach to *t*-way test sequence generation based on a Labeled Transition System (LTS) model that captures system behavior. An LTS model is similar to a finite state machine where the sequencing constraints are implicitly encoded in the state transitions. However, as an operational model, LTS is at a very low level of abstraction and requires a lot of details on how a system operates in terms of states and transitions. As a result, LTS models are seldom available in practice.

In this paper we develop a notation for specifying sequencing constraints and present a *t*-way test sequence generation algorithm that handles constraints specified in this notation. Our notation adopts an event-oriented framework. It defines a small set of basic operators that capture several fundamental orderings that could happen between two events. These operators can be nested, if necessary, to specify the sequencing behavior between multiple events. Our notation is at a higher level of abstraction than an operational model such as LTS. Also we believe that since we deal with sequences of events, an event-oriented notation is more intuitive than a state machine-based notation.

Our test sequence generation algorithm employs a greedy strategy in which each test sequence is generated such that a maximal number of *t*-way target sequences can be covered in the sequence. A *t*-way target sequence is a sequence of *t* events that could be covered in the given order, i.e., the order in which they appear in the sequence. Each test sequence is generated in two phases, including the starting phase and the extension phase. In the starting phase, we generate a starting sequence that is guaranteed to cover at least one target sequence. In the extension phase, we keep extending the test sequence until no extension is possible. At each extension, we append to the test sequence an event that covers the most *t*-way target sequences that are yet to be covered.

We report a case study in which we applied our approach to a communication protocol, i.e., IEEE 11073-20601 [10]. This protocol is used to exchange data between Personal Health Devices (PHDs), e.g., smart scales, and computing devices, e.g., desktop computers. We identified a set of 9 sequencing constraints and wrote them using our notation.

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00059

We generated a set of 24 test sequences that achieve 2-way sequence coverage while satisfying all the sequencing constraints. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for this protocol. However, while our algorithm allows us to generate *t*-way test sequence set, it is computationally expensive. For example, it takes 5 and half hours for generating test sequences with lengths up to 9, and 2 days with lengths up to 10. While this is partly due to the nature of the problem, we believe there are opportunities for optimization which we will explore in our future work.

We focus on positive testing in this paper. That is, we generate test sequences that satisfy all the sequencing constraints. However, sequencing constraints are also useful for negative testing. For example, test sequences could be generated for negative testing that violate constraints in a systematic manner.

The remainder of the paper is organized as follows. Section II explains our basic idea with a motivating example. Section III presents the syntax and semantics of our notation to specify sequencing constraints. We also introduce two other types of constraints, i.e., repetition and length constraints, which are used to control test sequence length. Section IV first presents the definitions of *t*-way target and test sequences, and then describes our test generation algorithm that handles constraints. Section V presents the results of our case study on the PHD protocol, including the sequencing constraints identified and some statistics on the test sequences generated. Section VI discusses related work on *t*-way test sequence generation. Section VII provides concluding remarks and our plan for future work.

II. Motivating Example

In this section, we use File API as a motivating example to show the basic idea of our work, including how to model the sequencing behavior of a system in terms of events and constraints, and how to generate a set of test sequences for t-way sequence coverage.

A. Model Sequencing Behavior

A sequencing model M = <E, C> consists of two components: (1) E: a set of events that could be exercised in a system execution; (2) C: a set of constraints that restrict the occurrences of these events in a system execution.

In File API, there are four major file operations, including *open*, *close*, *read*, and *write*. In the sequencing model, each of these operations is modeled as an event. Thus, E = {*open*, *close*, *read*, *write*}.

Based on the semantics of file operations, the following constraints can be identified in terms of the order in which these events could be exercised. These constraints are referred to as sequencing constraints.

(1) The first event of a test sequence must be *open*.

(2) The file must be open before *read*, *write,* or *close*.

(3) The last event of a test sequence must be *close*.

We introduce an event-oriented notation to model the above constraints. To model the first constraint, we specify that the *open* event must happen before all the other three events. More precisely, in a test sequence, whenever there is a *read*, *write*, or *close* event *e*, there must exist an *open* event that is exercised before *e*. To model the second constraint, we specify that *open* must happen before *read*, *write* or *close* and there shall be no *close* in between. To model the third constraint, we specify that the *close* event must happen after all other three events. More precisely, in a test sequence, whenever there is an *open*, *read*, or *write* event *e*, there must exist a *close* event that is exercised after *e*.

In addition to sequencing constraints, we introduce two other types of constraint, namely repetition and length constraints, to control the length of a test sequence. A repetition constraint specifies the number of times a certain event could be repeated in a test sequence. For example, we could specify that *open*/*close* could only occur once in a test sequence. A length constraint specifies the minimum and/or maximum length of a test sequence.

In Section III, we introduce a formal notation to specify the three types of constraints in detail.

B. Test Sequence Generation

After a sequencing model is specified, a test sequence set can be generated to achieve *t*-way sequence coverage. Recall that *t*-way sequence coverage requires that every *t*-way (target) sequence, i.e., every sequence of *t* events that could be exercised in the given order, consecutively or not, be exercised so by at least one test sequence.

For example, for File API, the set of 2-way sequences is {<*open*, *open*>, <*open*, *close*>, ..., <*read*, *write*>, ..., <*write*, *write*>} (the set size = 4^2 = 16). Note that the existence of constraints may make some of these sequences *uncoverable*, i.e., they cannot be covered by any test sequence that satisfies all the constraints. In this paper, we use <e_1, e_2, ...> to represent target sequences, and use [e_1, e_2, ...] to represent test sequences. A 2-way target sequence <e_1, e_2> is covered by a test sequence in the form of [..., e_1, ..., e_2, ...].

If we specify some repetition constraints such that no event could be repeated in a test sequence, in addition to the three sequencing constraints mentioned earlier, there is a total of seven 2-way target sequences {<*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*>, <*write*, *read*>}. A greedy algorithm can be used to generate a 2-way test sequence set such that all the test sequences satisfy all the constraints, and every target sequence is covered by at least one test sequence.

The details of the greedy algorithm are presented in Section IV. The following example illustrates the basic idea of the algorithm:

1. We first construct a starting sequence that covers at least one target sequence. We begin with an empty test sequence []. The only possible event that could be added is *open* due to sequencing constraint (1). The resulting sequence is [*open*].

2. We further extend [*open*]. There are three possible choices: [*open*, *read*]/[*open*, *write*]/[*open*, *close*]. Note that [*open*, *open*] is not allowed due to the repetition constraint.

All of the three choices cover one target sequence. We choose [*open*, *read*] as our starting sequence.

3. Now we try to extend the starting sequence. We append to the sequence one event at a time. We first append *write*, followed by *close*, as these two events allow the most target sequences to be covered. The resulting sequence is [*open*, *read*, *write*, *close*], which cannot be further extended.

4. We repeat the above process to generate additional test sequences until all the remaining target sequences are covered.

The final 2-way test sequence set consists of two test sequences {[*open*, *read*, *write*, *close*], [*open*, *write*, *read*, *close*]}. These two sequences satisfy all the constraints and cover all of the seven 2-way target sequences.

III. Notation for Constraint Specification

Our notation supports three types of constraints, including repetition, length and sequencing constraints. Recall that repetition and length constraints are used to control the length of a test sequence.

First, we introduce the syntax and semantics of repetition and length constraints:

- A repetition constraint is in the form of "$e.\# \leq r$", denoting that, in a test sequence, an event e could occur no more than r times. A default repetition constraint can be specified in the form of "$\# \leq r$", denoting that no event could occur for more than r times. The default constraint can be overridden by an event-specific constraint.
- A length constraint is in the form of "TOTAL_LEN ≥ *min*" or "TOTAL_LEN ≤ *max*", denoting that, the total length of a test sequence should be greater than or equal to *min*, or/and smaller than or equal to *max*.

By default, "$\# \leq 1$" is given to ensure the termination of test sequence generation. That is, by default, each event is only allowed to appear once in a test sequence.

The rest of this section is focused on the syntax and semantics of sequencing constraints.

A. Syntax of Sequencing Constraints

The syntax of sequencing constraints is specified in BNF (Backus–Naur form) as shown in Fig. 1.

```
<sequencing constraint> ::= <sequencing expression>
      | <sequencing constraint> ∨ <sequencing constraint>
      | <sequencing constraint> ∧ <sequencing constraint>
      | (<sequencing constraint>)
<sequencing expression> ::=
      <sequencing expression> <general sequencing operator> <events>
      | <events> <general sequencing operator> <events>
      | <events> <immediate sequencing operator> <events>
<events> ::= <event> | <event set>
      | <always sequencing operator> <event>
      | <always sequencing operator> <event set>
<always sequencing operator> ::= "_"
<immediate sequencing operator> ::= "*–" | "–*" | "~"
<general sequencing operator> ::= "*…" | "…*" | "·~·"
```

Figure 1. BNF of sequencing constraints

There are two types of operators: Boolean and sequencing operators. Sequencing operators can be divided into three groups: immediate operators, general operators, and always operator.

Note that <event set> is a set of events. That is, we allow event sets, as well as individual events, in a constraint expression. In this paper, we will use the notation of {e_1, e_2, …, e_n} to denote an event set, where e_1, e_2, …, e_n are individual events. The reason why this is allowed is explained in Section III.B.

A sequencing constraint can be derived from this syntax as shown in Fig. 2.

```
<sequencing constraint>
=> <sequencing expression>
=> <sequencing expression> <general sequencing operator> <events>
=> <events> <general sequencing operator> <events> <general sequencing
      operator> <events>
=> <always sequencing operator> <event> <general sequencing operator>
      <event> <general sequencing operator> <event set>
=> open ·~· close ···* {read, write, close}
```

Figure 2. Derivation of a sequencing constraint from BNF

Note that the precedence of the operators is defined from highest to lowest as follows: unary sequencing operator, binary sequencing operators, (), ∧ and ∨.

B. Semantics of Sequencing Operators

TABLE I. Informal Explanation of Sequencing Operators

Sequencing Operator	Explanation
_e (or <u>e</u>)	e **always** happens
e_1 *– e_2	If e_1 happens, then e_2 **must immediately happen after** e_1
e_1 –* e_2	If e_2 happens, then e_1 **must immediately happen before** e_2
e_1 ~ e_2	e_2 **never immediately happens after** e_1 (or e1 **never immediately happens before** e_2)
e_1 *··· e_2	If e_1 happens, then e_2 **must happen after** e_1, **but not necessarily immediately happen after** e_1
e_1 ···* e_2	If e_2 happens, then e_1 **must happen before** e_2, **but not necessarily immediately happen before** e_2
e_1 ·~· e_2	e_1 **never happens before** e_2 (or e_2 **never happens after** e_1)

TABLE I lists all the sequencing operators in our notation and provides an informal explanation of each operator. We make the following notes about the symbols used to represent the operator:

(a) _: This indicates that the event always happens. (In this paper, this operator is shown as an underline, e.g., <u>e</u>.)

(b) –/···: Both indicate the left event happens before the right event. However, – requires that the two events are next to each other, whereas ··· does not.

(c) ~/·~·: Both indicate the left event never happens before the right event. However, ~ only requires that the two events do not happen next to each other, whereas ·~· requires that the right event cannot happen one after the left event.

(d) *: It indicates that the constraint applies only if the left (or right) event exists, if it appears left (or right) to the operator.

Recall that we allow a sequencing operator to be applied to a set of events. For example, "E1 *— E2" denotes that, immediately after any event in set E1, an event in set E2 must happen. The reason why this is necessary is that Boolean operators on sequencing constraints with individual events cannot specify some constraints. For example, "e_1 *— {e_2, e_3}" ≠ "e_1 *— e_2 ∨ e_1 *— e_3". Given a test sequence [...e_1, e_2, ..., e_1, e_3, ...] containing two occurrences of event e_1, "e_1 *— e_2" is not satisfied on the second occurrence of e_1, while "e_1 *— e_3" is not satisfied on the first occurrence of e_1. Thus, none of the two constraints are satisfied. However, the constraint "an event belongs to {e_2, e_3} must happen after e_1" denoted by "e_1 *— {e_2, e_3}" is satisfied on both occurrences of e_1.

In the following, we use an automaton to formally define the semantics of each sequencing operator. (Note that in each automaton, ¬ e indicates any event other event e; ∀e indicates any event; e_1 ∨ e_2 indicates e_1 or e_2.)

1. Always sequencing operator

(1) $\underline{e}$

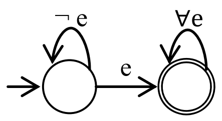

Figure 3. Semantics of "$\underline{e}$"

Fig. 3 shows that, given an input sequence, if an occurrence of event e exists, the sequence should be accepted. Otherwise, it is rejected.

2. Immediate sequencing operators

(2) e_1 *— e_2

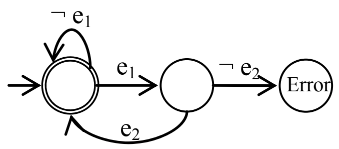

Figure 4. Semantics of "e_1 *— e_2"

Fig. 4 shows that given an input sequence, if every occurrence of event e_1 is immediately followed by an occurrence of event e_2, the sequence should be accepted. Otherwise, it is rejected.

(3) e_1 —* e_2

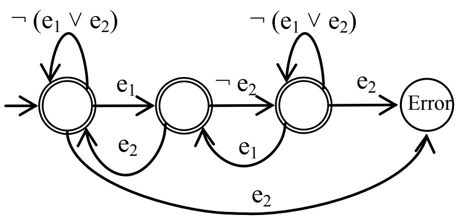

Figure 5. Semantics of "e_1 —* e_2"

Fig. 5 shows that given an input sequence, if any occurrence of event e_2 exists that is NOT immediately after an occurrence of event e_1, the sequence should be rejected. Otherwise, it is accepted.

(4) e_1 ~ e_2

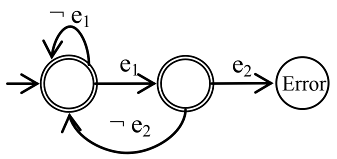

Figure 6. Semantics of "e_1 ~ e_2"

Fig. 6 shows that, given an input sequence, if any occurrence of event e_2 exists immediately after an occurrence of event e_1, the sequence should be rejected. Otherwise, it is accepted.

3. General sequencing operators

The automaton of each general operator is similar to the automaton of the corresponding immediate operator, but in general simpler. This is because the semantics of the immediate operators are stricter, except that ·~· is stricter than ~.

(5) e_1 *··· e_2

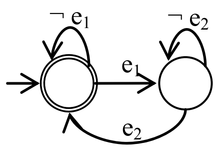

Figure 7. Semantics of "e_1 *··· e_2"

(6) e_1 ···* e_2

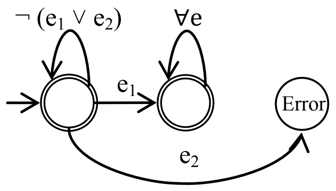

Figure 8. Semantics of "e_1 ···* e_2"

(7) e_1 ·~· e_2

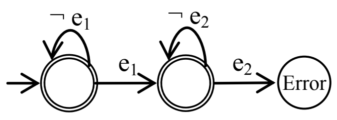

Figure 9. Semantics of "e_1 ·~· e_2"

General sequencing operators can be nested in our notation. The semantics of a nested expression "<sequencing expression> <general sequencing operator> <events>", where <sequencing expression> is the nesting expression, can be defined in a recursive manner. As an example, consider "B ···* e_3", where B = "$\underline{e_1}$ ·~· e_2". This nested expression can be written as "$\underline{e_1}$ ·~· e_2 ···* e_3". It denotes that if event e_3 happens, B must be satisfied by a subsequence before e_3 in which "e_1 always happens, and e_2 never happens after e_1". In other words, before each occurrence of e_3, there must be an occurrence of e_1 to satisfy B, so that e_2 never happens between this occurrence of e_1 and the occurrence of e_3.

Recall that a sequencing operator may involve an event set E. The semantics of each operator with a set of events can be specified using the corresponding automaton with individual events, except that we need to replace "event e" with "any event in E".

C. Example

We can use our constraint notation to represent the sequencing constraints identified in Section II as follows:

(1) "*open* ···* {*close, read, write*}": This constraint says that, if there exists an event *close*, *read* or *write*, there must exist an *open* event before this event. This ensures that *open* is the first event.

(2) "*open* ·~· *close* ···* {*read, write, close*}": This constraint says that if there exists an occurrence of event *read*, *write*, or *close*, there **must** exist an occurrence of event *open* **before** it, and the subsequence separated by the two occurrences satisfies "after this occurrence of *open*, before this occurrence of *read*, *write*, or *close*, an occurrence of *close* **never** exist". This ensures that the file is open before it is read, written, or closed.

(3) "{*open*, *read*, *write*} *··· *close*": This constraint says that, after there exists an *open*, *read* or *write* event, a *close* event must exist. This ensures that *close* is the last event.

IV. Approach

A. Basic Concepts

Informally, a target sequence is a sequence of events that needs to be covered. A test sequence is a sequence of events that can be executed by the subject system. Target sequences are covered by test sequences to satisfy *t*-way sequence coverage. In other words, target sequences are test requirements, i.e., entities that must be covered to achieve *t*-way sequence coverage, while test sequences are test cases that cover test requirements. T-way sequence coverage requires that for every target sequence of *t* events, if they could be exercised in the given order, they must be exercised so in at least one test sequence.

In the following, we formalize these concepts from the perspective of test sequence generation, without considering the semantics of the subject system. Let M = <E, C> be the sequencing model of the subject system.

Definition 1. A sequence Q of events is *valid* if it satisfies all the constraints in C; otherwise it is invalid.

In this paper we focus on positive testing. Thus, every test sequence must be a valid sequence. Also, every valid sequence can be used as a test sequence.

Definition 2. A sequence Q of events is *extendable* if it is a proper prefix of another sequence of events that is valid.

Note that an extendable sequence itself may or may not be valid.

Definition 3. A sequence Q of events *covers* another sequence Q' of events if all the events in Q' appear in Q in the same order as they appear in Q'.

In the above definition, it is important to note that the events in Q' do not have to appear consecutively in Q. For example, a partial test sequence [*open*, *read*, *write*] covers three 2-way sequences: <*open*, *read*>, <*open*, *write*> and <*read*, *write*>.

Definition 4. A *t*-way target sequence Q is a *t*-way sequence that can be covered by at least one test sequence.

Note that not every *t*-way sequence is a target sequence. Consider the File API example. If a repetition constraint requires that no event can be repeated, then 2-way sequences <*open*, *open*> and <*close*, *open*> cannot be covered by any test sequence. Thus, these sequences are not 2-way target sequences.

Definition 5. Let Π be the set of all the *t*-way target sequences. A *t*-way test sequence set Σ is a set of test sequences such that for $\forall \pi \in \Pi$, $\exists Q \in \Sigma$ such that Q covers π.

Considering the motivating example, a set of two test sequences {[*open*, *read*, *write*, *close*], [*open*, *write*, *read*, *close*]} covers all 2-way target sequences <*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*> and <*write*, *read*>.

Note that the above definitions are similar to our earlier work in [7], which is however based on LTS.

B. Main Idea

```
Input: (a) A sequencing model M = (E, C), where E is a set of events and C
is a set of constraints, and (b) a test strength t
Output: A t-way test sequence set Σ
{
   // Step 1: target sequence (candidate) generation
 1. let Π be {π = <e1, e2, …, et> | ei ∈ E}
   // Step 2: test sequence generation
 2. let Σ be an empty set
 3. while (Π is not empty) {
     // Step 2.1: starting phase
 4.   create a starting test sequence Q such that (a) Q covers at least one
            target sequence in Π; and (b) Q is valid or extendable
 5.   if (Q cannot be created)
 6.     break
 7.   remove from Π the target sequences covered by Q
     // Step 2.2: extension phase
 8.   while (Q is extendable) {
 9.     append an event e in E to Q such that (a) Q.e covers the most target
              sequences in Π; and (b) Q.e is valid or extendable
 10.    Q = Q.e
 11.    remove from Π the target sequences covered by Q
 12.  }
 13.  add Q into Σ
 14. }
 15. return Σ
}
```

Figure 10. Algorithm GenTestSeqs

Fig. 10 shows our test generation algorithm. The algorithm consists of three major steps. The first step is to generate target sequence candidates. Note that not every sequence in Π is a target sequence, as some sequences in Π may not be covered by any test sequence. As discussed later, the second step guarantees that all the target sequences in Π will be covered. Thus, after the second step, the remaining sequences in Π cannot be covered by any test sequence and are not target sequences. We could check whether every sequence in Π is a target sequence and remove those that are not prior to the second step. This, however, is expensive and redundant.

In the second step, we generate test sequences to cover all the target sequences. We first create a starting test sequence Q to cover at least one remaining target sequence. A starting test sequence must be valid or extendable. This is necessary to ensure termination. If such a test sequence cannot be created, all the target sequences in Π have already been covered, and the algorithm terminates. Otherwise, we extend Q by appending events one by one. Each time we select an event that covers the most target sequences in Π. When no event can be appended to Q, Q becomes a *complete* (valid and not-extendable) test sequence, and we add Q into the resulting test set and create another starting test sequence. We continue to do so until we cannot find a starting sequence that covers at least one sequence in Π.

We call the phase of creating a starting test sequence as a starting phase in Line 4 - 7, and the phase of extending a test sequence to be complete as an extension phase in Line 8 - 13.

C. *Validity and Extensibility Check*

In this part we discuss how to check if a test sequence is valid and if a test sequence is extendable. These are two important checks performed in our algorithm.

1. Validity check: Recall that there are three types of constraints, sequencing, repetition, and length constraints.

Given a test sequence Q, we first check whether it satisfies all the repetition and length constraints, which is accomplished by counting its number of events. Next we check whether it satisfies all the sequencing Constraints, which is more complicated and is described as follows.

As indicated by the BNF grammar of Section III, there are two types of constraints for solving: basic and nested.

(1) A basic expression “*e* <sequencing operator> *e*” only involves two events (or event sets) in sequence, such as “e_1 *− e_2”. Based on basic temporal logic (as corresponding automaton), the basic expression is true on Q if and only if Q is accepted by our automaton. Note that, our Sequencing Constraint Solver is only applicable to test sequence, i.e., not target sequences whose validity need to be checked differently.

(2) A nested expression “B <sequencing operator> *e*” involves more than two events in sequence, since B is another sequencing expression, such as “$\underline{e_1}$ ·~· e_2 *··· e_3”, (i.e., “B *··· e_3”, B = “$\underline{e_1}$ ·~· e_2”). Automatons will be recursively called by the nested structure. The Boolean result of the nested expression “B *··· e_3” on Q is decided by “if B is true on a subsequence of Q, whether e_3 happens after the subsequence”. Thus, the global Boolean result is that “if ‘e_1 always happens and e_2 never happens after e_1’ is true, whether e_3 happens after e_1”.

2. Extensibility check: Fig. 11 shows our algorithm for checking whether a sequence is extensible. The algorithm employs a recursive DFS (Depth-First Search) strategy. Note that in order to prevent infinite extension, we set a default repetition constraint which requires every event be repeated no more than *t* times, where *t* is the coverage strength, if the user does not specify any length constraint to restrict the maximum length of a sequence. Thus, a maximum length could always be derived from the repetition and length constraint.

```
Boolean isExtendable(Q, E, C)
{
  let max_length be the maximum length implied by all the repetition and
      length constraints
  **if** (Q.length >= max_length)
    return false
  **for** (each event *e* in *E*) {
    set Q' to be Q.*e*
    **if** (isValid(Q', C))
      return true
    **else if** (isExtendable(Q', E, C))
      return true
  }
  return false
}
```

Figure 11. Extensibility Check Algorithm

D. *Test Sequence Generation*

In this part, we discuss two main challenges of our generation approach shown in Fig. 10.

The first challenge of our generation approach is that, due to the limitation of our automatons which are only available for consecutive sequence, we cannot directly check the validity of target sequences.

As indicated in Line 1, we enumerate all possible permutations (with repetition) of any *t* events as *t*-way target sequence candidates. The same event could be exercised for up to *t* times in a permutation. Recall that some of these candidates cannot be covered by any test sequence, while others can be covered.

Our solution is to remove covered *t*-way sequences from the set of candidates during test sequence generation. After the generation finished, we consider the remaining uncovered *t*-way sequence candidates as invalid. The reasons why our solution works are as follows.

(1) Covered target sequences must be valid: According to our definition of valid sequence, and our previous research on constraint handling [9], all subsequences covered by a test sequence are valid. In other words, an invalid *t*-way sequence cannot be covered by any test sequence.

(2) Valid target sequences must be covered: For each starting phase, it ensures to cover at least one remaining *t*-way sequence candidate, until no such starting sequence can be created. So, before the break in Line 6, all valid *t*-way sequence candidates must have been covered by test sequences.

The second challenge of our generation approach is to create a starting sequence, i.e., a valid or extendable test sequence that covers at least one remaining *t*-way target sequence, within a reasonable time.

As indicated in Line 4, in order to ensure termination of test sequence generation, we create a starting test sequence that covers at least one target sequence in Π. Our solution is to adopt a BFS (Breadth-First Search) strategy as in Fig. 12.

Note that, validity and extensibility check can only guarantee to generate a complete test sequence, not for coverage, which may not cover any remaining target sequence.

```
Input: (a) A sequencing model M = (E, C), where E is a set of events and C
is a set of constraints, (b) a test strength t, and (c) a set of remaining target
sequence candidates Π
Output: A starting test sequence Q
{
 // Initialize a queue of starting sequence candidates U
  let U be a queue consisting of all the event sequences of length t
  while (U is not empty) {
    remove the first sequence Q from U
    if (Q covers at least one target sequence in Π) {
      if (Q is valid or extendable)
        return Q
    }
    else if (Q is extendable) {
      for (each event e in E)
        append e to Q and add it to the end of U
    }
  }
  return null
}
```

Figure 12. Algorithm for creating a Starting Test Sequence

V. Case Study

In this section, we apply our test sequence generation framework to the IEEE 11073-20601 protocol (Optimized Exchange Protocol) [10]. As a core component in the standards family of IEEE 11073, this protocol defines a communication model that allows PHDs (Personal Healthcare Devices) to exchange data with computing devices like mobile phones, set-top boxes, and personal computers.

A. Overview of the Protocol

In IEEE 11073, there are two types of devices, agent and manager devices. Agents are personal healthcare devices that are used to obtain measured health data from the user. Examples of agents include blood pressure monitors, weighing scales and blood glucose monitors. Managers manage and process the data collected by agents. Examples of managers include mobile phones, set-top boxes and PCs.

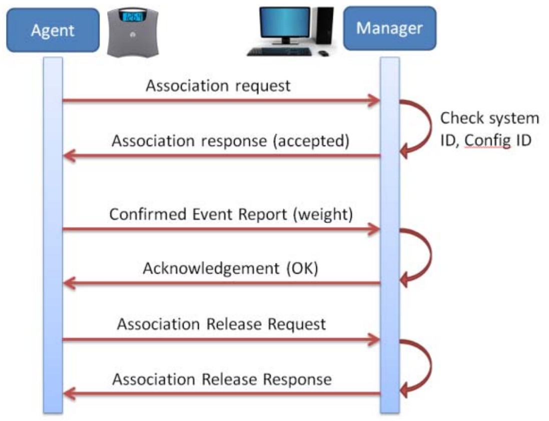
Agent
Manager
Association request
Check system ID, Config ID
Association response (accepted)
Confirmed Event Report (weight)
Acknowledgement (OK)
Association Release Request
Association Release Response

Figure 13. An example scenario of Data Exchange

We reuse an example scenario from our earlier work [8] to illustrate how an agent exchanges data with a manager. In the scenario, the agent device is a weighting scale. It sends an *Association request* to the manager. The association request contains the weighting scale's configuration information, e.g., system ID, protocol version number. If the manager recognizes the agent, it accepts the association request and sends to the agent an *Association acceptance* message. At this point, the two devices are ready to exchange actual data. Next the agent sends measurement data, e.g., weight information, to the manager using a *Confirmed Event Report* message. The manager successfully receives the Confirmed Event Report and sends back the acknowledgement. At the end of this scenario the agent requests to release the association with an *Association release request* message, and the manager releases the association and sends back to the agent an *Association release response* message.

In this case study, we identify sequencing constraints that the protocol imposes on the communication behavior. We specify these constraints using the notation developed in Section III and generate *t*-way test sequences that satisfy these constraints. These test sequences can be used to perform conformance testing of an implementation of the protocol, e.g., Antidote [11].

B. Sequencing Constraints

We identify constraints from the manager's perspective. Constraints can be similarly identified from the agent's perspective. In particular, as participants of the same protocol, agent and manager exhibit to a large extent symmetrical behavior, in terms that a send event on one side corresponds to a receive event on the other side.

The events on the manager side can be divided into three groups, based on their source and destination:

- Event beginning with REQ – There is a single request event, *REQ_assoc_rel*, sent from the application software interface, and it is triggered and handled inside the manager.

- Events beginning with Rx – These events are requests sent from the agent to the manager. They include *Rx_assoc_rel_req, Rx_assoc_rel_rsp, Rx_assoc_req, Rx_config_event_report_req*.

- Events beginning with Tx – These events are responses sent from the manager to the agent. They include *Tx_assoc_rel_req, Tx_assoc_rel_rsp, Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config, Tx_config_event_report_rsp_accepted_config, Tx_config_event_report_rsp_unsupported_config.*

Thus, there is a total of 12 events including 1 REQ, 4 Rx, and 7 Tx events for the manager. Note that we ignore the abort events which can happen anywhere, since we focus on positive testing.

Alternatively, the events can be divided into three groups, based on their functional areas.

- Events that establish association:

Rx_assoc_req and its 3 possible responses *Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*;

- Events that release association:

Rx_assoc_rel_req and its response *Tx_assoc_rel_rsp*; *REQ_assoc_rel*, *Tx_assoc_rel_req* and its response *Rx_assoc_rel_rsp*;

- Events that check configuration:

Rx_config_event_report_req and its 2 possible responses *Tx_config_event_report_rsp_accepted_config, Tx_config_event_report_rsp_unsupported_config*.

In our study, we first identify constraints from each functional group separately and then put them together, e.g., using nested expressions. The final constraints are shown in Fig. 14.

1. *Rx_assoc_req* ···* {all the events except *Rx_assoc_req*}
2. {all other events except the right three events} *··· {*Tx_assoc_rel_rsp, Rx_assoc_rel_rsp, Tx_assoc_rsp_rejected*}
3. *Rx_assoc_rel_req* − *Tx_assoc_rel_rsp*
4. *Rx_assoc_req* − {*Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*}
5. *Rx_config_event_report_req* − {*Tx_config_event_report_rsp_accepted_config*, *Tx_config_event_report_rsp_unsupported_config*}
6. (*Tx_assoc_rel_req* ···* *Rx_assoc_rel_rsp*) ∧ (*Tx_assoc_rel_req* ·~· {*Tx_assoc_rel_rsp*, *Rx_assoc_rel_rsp*} *··· *Rx_assoc_rel_rsp*)
7. {*Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*} ·~· {*Tx_assoc_rel_rsp*, *Rx_assoc_rel_rsp*} ···* {*Rx_assoc_rel_req, REQ_assoc_rel*}
8. (*REQ_assoc_rel* − *Tx_assoc_rel_req*) ∧ (*Tx_assoc_rel_req* *− {*Rx_assoc_rel_rsp, Rx_assoc_rel_req*}) ∧ (*Tx_assoc_rel_req* −* *Rx_assoc_rel_rsp*)
9. {*Tx_assoc_rsp_accepted_unknown_config, Tx_config_event_report_rsp_unsupported_config*} ·~· {*Rx_assoc_rel_req*, *REQ_assoc_rel*} ···* *Rx_config_event_report_req*

Figure 14. All 9 sequencing constraints of PHD manager model

Constraint 1. *Rx_assoc_req* is the first event that must happen before all other events. This event requests association to be established.

Constraint 2. *Tx_assoc_rel_rsp, Rx_assoc_rel_rsp, Tx_assoc_rsp_rejected* are the last events that must happen after all other events. These events indicate that association has been released or rejected.

Constraints 3-5. For convenience, we write "e_1 −* e_2 ∧ e_1 *− e_2" in its abbreviated form "e_1 − e_2". Based on the protocol semantics, after the manager receives a request, it must immediately transmit an event as response.

Constraint 6. To maintain causal semantics, when a response event happens, its corresponding request event must happen before it. However, after a request event occurs, a response event may not always happen, e.g., due to disconnection or disassociation.

In the PHD protocol, after a request is transmitted, an event of its possible response may not be received when association has already been released by other events, which is indicated by events *Tx_assoc_rel_rsp* or *Rx_assoc_rel_rsp*. In other words, if these two events don't happen after *Tx_assoc_rel_req*, then *Rx_assoc_rel_rsp* must happen in some time.

Constraint 7. We have two request events *Rx_assoc_rel_req* and *REQ_assoc_rel* to release association from agent and manager side. These two events can only happen when the association is accepted and not yet released.

Constraint 8. When the manager triggers *REQ_assoc_rel*, it will immediately transmit a release request, and then busy wait until it receives either the release response or another release request from the agent. The constraint restricts that the manager must not finish association release until the agent agrees.

Constraint 9. Similar to constraint 7, *Rx_config_event_report_req*, which receives a new configuration from the agent, can only happen after the previous configuration is checked to be unknown or unsupported and no release request has happened.

One benefit of our notation is that it allows incremental specification. That is, we do not require all the constraints be specified up front. Instead, we can begin with several constraints, generate test sequences that satisfy these constraints, and then check whether these sequences are as expected. If not, we can add more constraints. This can be repeated for multiple times until we capture all the constraints.

C. Test Sequence Generation Results

The experimental environment is set up as the following: OS: Windows 7 64bits, CPU: Intel Dual-Core i5 2.5GHz, Memory: 8 GB DDR3, SDK: Java SE 1.7.

We use the 9 sequencing constraints in Fig. 14 with different repetition and length constraints in TABLE II to generate test sequences that achieve 2-way sequence coverage.

TABLE II. RESULTS OF 2-WAY TEST SEQUENCE GENERATION

Rep cons	Len cons	# of target seqs	Gen Time(sec)	# of test seqs	test seq length		
					min	avg	max
≤ 1	≤ 6	36	3.9	7	2	4.6	6
≤ 1	≤ 7	45	26.7	9	2	5.1	7
≤ 1	≤ 8	45	155.5	9	2	5.1	7
≤ 2	≤ 6	61	13.1	15	4	5.7	6
≤ 2	≤ 7	79	157.1	16	4	6.4	7
≤ 2	≤ 8	105	1789.5	26	4	7.4	8
≤ 2	≤ 9	123	19802.9	24	4	8.2	9
≤ 2	≤ 10	135	206191.9	24	4	8.4	10

TABLE II shows that the test generation time grows quickly as the maximum length of a test sequence increases. We believe our test generation algorithm has a lot of room for optimization, which will be explored in our future work.

Note that after we set the maximum repetition and maximum length constraint, test sequences may not grow up to the maximum length. For example, in TABLE II, for the third experiment, where each event can only appear once and the length limit is 8, the maximum length of a test sequence we generate is 7. The reason is that sequencing and repetition constraints may interact to reduce the maximal length of a test sequence.

Also note that the number of target sequences increases as we relax the repetition and length constraints. Since the number of events is 12, the test strength is 2, the number of possible 2-way sequences is 12^2 = 144. Some of them cannot be covered due to repetition, length, and sequencing constraints.

VI. Related Work

Combinatorial testing has been an active area of research [6]. However, most work has focused on t-way test data generation [12]. In this section, we focus our discussion on *t*-way sequence generation that supports constraints.

There exist many *t*-way test sequence generation approaches supporting constraints. However, some of them lack the capability to specify all possible constraints for real-life systems, while others require a low-level specification of constraints such as dependency graph or state transition diagram.

Kuhn et al. [4] presented an approach to generating *t*-way SCAs. Their approach require each event to appear exactly once in a test sequence. Thus the length of each test sequence is fixed, which equals the number of events. It supports one type of constraint on sequence "*x*..*y*", which means that no test sequence should contain *x* and *y* in the given order. This is similar to our notation "*x* ·~· *y*". This notation cannot specify constraints involving more than two events. For example, it cannot specify that some event must or never happen between two events. Furthermore, there are certain types of constraints between two events that cannot be specified by this notation. For example, consider the constraint in our notation, "*x* ···* *y*", meaning that if *y* happens, *x* must happen before *y*. This constraint cannot be specified using the notation in [4] to prevent sequence "*y*..*x*". This is because a test sequence in the form of "[… *x* … *y* … *x* … *y* …]" satisfies this constraint, but *x* and *y* appear in different orders in the same sequence.

Farchi et al. [13] developed an approach to generating test sets that satisfy ordered and unordered interaction coverage. Ordered restrictions can be considered as a type of sequencing constraints. For example, the ordered restriction excluding a case "Read.comesBefore(Open)" to prevent <Read, Open> from generation. This restriction is similar to the notation in [4], and thus has similar limitations as mentioned earlier.

Several approaches have been reported that use a graph model to represent system behavior from which *t*-way test sequences are generated. Wang et al. [14] presented a pairwise test sequence generation approach for web applications. Their approach is based on a graph model called navigation graph that captures the navigation structure of a web application. Rahman et al. [15] presented a test sequence generation approach using simulated annealing. Their approach is based on a state transition diagram that models the system behavior. Yu et al. [7][8] presented several algorithms that generate *t*-way test sequences from LTS models. In these approaches, sequencing constraints are implicitly encoded in the graph model. Compared to our notation, the graph models used in these approaches are at a lower level of abstraction and require a lot of operational details that may not be readily available in practice.

Kruse et al. [16] suggested that temporal logic formulas, e.g., Linear Temporal Logic (LTL) [17], Computational Tree Logic (CTL) [18], and modal μ-calculus [19], can be used to express sequencing constraints. They used LTL for dependency rules (i.e., sequencing constraints) and CTL for generation rules (i.e., strength *t*, repetition and length constraints). Temporal logic formulas are powerful in terms of the different types of property they could be used to express. However, these notations have a complex semantic model, and have found limited use in practice. For example, both LTL and CTL have a state-based semantic model. In theory, any state-based property can be specified using events, and vice versa. However, the notion of state is more difficult to grasp than that of event. This is because unlike events, states are not directly represented in a test sequence. Thus, in order to specify sequencing constraints, events must be translated into states. This translation can be difficult due to the fact that states can be defined at different levels of abstraction and thus the mapping between states and events may not be a simple one-to-one relation.

Dwyer et al. [20] developed a system of property specification patterns to specify properties that are commonly encountered in practice. Our work is different in that we define a minimal set of basic operators, each of which captures a fundamental relationship between events. Complex properties can be specified using these basic operators. The work in [20] is complementary with ours in that similar patterns can also be identified to facilitate the use of our notation in practice.

VII. Conclusion and Future Work

There seems to be a significant amount of interests on *t*-way sequence testing in both academia and industry. However, progress is still lacking. In this paper we present an approach to handling sequencing constraints, which we believe is a key technical challenge in *t*-way test sequence generation but has not been adequately addressed. Our approach consists of an event-oriented notation for expressing sequencing constraints and a greedy algorithm for generating test sequences that achieve *t*-way coverage while ensuring that all the constraints are satisfied. We applied our approach to a real-life communication protocol. Our experience suggests that our notation is more intuitive to use and can capture important sequencing constraints for this protocol. However, our test generation algorithm seems to be time consuming. This work is part of our larger and ongoing effort to make *t*-way sequencing testing practically useful.

In the future, we will continue our work in the following major directions. First, we want to optimize the performance of our test sequence generation algorithms. For example, there seems to be quite some redundant computations in the generation process. We plan to explore ways to reduce such redundancy, e.g., by saving intermediate results. Second, we want to develop an algorithm to perform consistency check on constraints specified by the user. This is necessary because the user may specify constraints that contradict with each other. This consistency check can reject contradictory constraints prior to test generation and can also provide feedback to the user in terms of how to make corrections. Finally, we want to investigate the formal properties of our notation for sequencing constraints, in terms of what kind of constraints our notation can or cannot express. In particular, we want to check the possible equivalence relation between our notation and other notations such as LTS and LTL. For example, is it true that any properties that can be expressed using LTS or LTL can be expressed using our notation and vice versa?

ACKNOWLEDGMENT

This research is partly supported by two research grants (70NANB15H199, 70NANB18H207) from National Institute of Standards and Technology.

Disclaimer: We identify certain software products in this document, but such identification does not imply recommendation by the US National Institute of Standards and Technology or other agencies of the US government, nor does it imply that the products identified are necessarily the best available for the purpose.

REFERENCES

[1] A. Canny. "Interactive system testing: beyond GUI testing." In Proceedings of the 2018 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, p. 18.

[2] R.C. Bryce, S. Sampath, and A.M. Memon. "Developing a single model and test prioritization strategies for event-driven software." IEEE Transactions on Software Engineering 37, no. 1 (2011): 48-64.

[3] D.E. Simos, J. Bozic, B. Garn, M. Leithner, F. Duan, K. Kleine, Y. Lei, and F. Wotawa. "Testing TLS using planning-based combinatorial methods and execution framework." Software Quality Journal (2018): 1-27.

[4] D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, and Y. Lei. "Combinatorial methods for event sequence testing." In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 601-609.

[5] F. J. Daniels, and K. C. Tai. "Measuring the effectiveness of method test sequences derived from sequencing constraints." In Proceedings of the 1999 Technology of Object-Oriented Languages and Systems, pp. 74-83.

[6] C. Yilmaz, S. Fouche, M.B. Cohen, A. Porter, G. Demiroz, and U. Koc. "Moving forward with combinatorial interaction testing." Computer 47, no. 2 (2014): 37-45.

[7] L. Yu, Y. Lei, R.N. Kacker, D.R. Kuhn, and J. Lawrence. "Efficient algorithms for t-way test sequence generation." In 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 220-229.

[8] L. Yu, Y. Lei, R.N. Kacker, D.R. Kuhn, R.D. Sriram, and K. Brady. "A general conformance testing framework for IEEE 11073 PHD's communication model." In sixth International Conference on Pervasive Technologies Related to Assistive Environments (PETRA 2013), p. 12.

[9] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. "An efficient algorithm for constraint handling in combinatorial test generation." In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 242-251.

[10] J.H. Lim, C. Park, S.J. Park, and K.C. Lee, "ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device." In 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society. pp. 1161-1164.

[11] Antidote IEEE 11073-20601 stack library. [Online]. http://oss.signove.com/index.php/Antidote:_IEEE_11073-20601_stack

[12] K. Kleine, and D.E. Simos. "An efficient design and implementation of the In-Parameter-Order algorithm." Mathematics in Computer Science 12, no. 1 (2018): 51-67.

[13] E. Farchi, I. Segall, R. Tzoref-Brill, and A. Zlotnick. "Combinatorial testing with order requirements." In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), , pp. 118-127.

[14] W. Wang, S. Sampath, Y. Lei, and R.N. Kacker. "An interaction-based test sequence generation approach for testing web applications." In Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp. 209-218

[15] M. Rahman, R.R. Othman, R.B. Ahmad, and M.M. Rahman. "Event driven input sequence t-way test strategy using simulated annealing." In 2014 5th International Conference on Intelligent Systems, Modelling and Simulation (ISMS), pp. 663-667.

[16] P.M. Kruse, and J. Wegener. "Test sequence generation from classification trees." In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 539-548.

[17] A. Pnueli. "The temporal logic of programs." In 1977 18th Annual Symposium on Foundations of Computer Science, pp. 46-57..

[18] E.M. Clarke, and E.A. Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic." In 1981 Workshop on Logic of Programs, pp. 52-71.

[19] D. Kozen. "Results on the propositional μ-calculus." Theoretical computer science 27, no. 3 (1983): 333-354.

[20] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. "Patterns in property specifications for finite-state verification." In Proceedings of 1999 21st International Conference on Software Engineering (ICSE), pp. 411-420.