MODMED

WP4/D1: Test assessment - preliminary study and tool prototype

Version 1.0
March 14, 2018

MODMED (ANR-15-CE25-0010) 2015-2018

Author	Partner	
Yoann Blein	LIG	Author
Mohammad Ali Tabikh	LIG	Author
Yves Ledru	LIG	Author
Lydie du Bousquet	LIG	Reviewer
Roland Groz	LIG	Reviewer
Fabrice Bertrand	BlueOrtho	Reviewer
Arnaud Clère	MinMaxMedical	Reviewer

Contents

1 Introduction **4**
1.1 Motivation . . . 4
1.2 Coverage based on disjunctive form . . . 4
1.3 Contents of this deliverable . . . 4
1.4 Short paper : illustrative examples of the coverage approach . 5
1.5 Internship report . . . 5
1.6 Tooling . . . 6

2 Revised rewriting system for ParTraP **8**

3 FM 2018 paper **11**

4 Internship report of Mohammad Ali Tabik **19**

1 Introduction

1.1 Motivation

The MODMED project has designed a language for the expression of properties of traces of parametric events. Properties written in the language may be quite complex, and may include several subcases. Therefore, it is interesting to know which subcases are covered by a set of traces. In the case where these traces correspond to traces of system tests, it may reveal that some special cases were not covered by the tests and lead to the definition and execution of new test cases. The set of traces may also correspond to usage traces, collected when the system is deployed. In such a case, the coverage information allows to have a fine understanding of how the property is covered in practice.

1.2 Coverage based on disjunctive form

In the MODMED proposal, we considered several ways to address coverage of ParTraP properties. A first approach is to compile properties into monitors, written in Java, and to measure the coverage of Java code while evaluating the property on the trace or on the set of traces. This approach could not be experimented until now because the implementation of ParTraP used an interpreter and was not compiled into Java monitors. Recently, we have implemented a ParTraP compiler which produces Java monitors, so we should be able to experiment this first approach in the coming months.

Another approach is to decompose each ParTraP formula into subproperties, and check which ones of these properties are covered by the trace. In order to explore this approach, we studied the rewriting of the ParTraP property into a disjunctive form. We were then able to evaluate which disjuncts were actually verified by the traces. It is this approach that is discussed thoroughly in this deliverable.

The idea of measuring coverage of temporal properties based on Dwyer's patterns has been explored by Cabrera Castillos et al. [1]. They transform temporal properties into automata and measure the coverage of these automata by a test suite. Instead we proposed a rewriting system which keeps the expression of properties in the original ParTraP language. Hence the user does not need to master a different formalism to understand coverage information.

1.3 Contents of this deliverable

This deliverable is based on two documents produced by the MODMED participants:

- The internship report of Mohammad Ali Tabikh for his Master's degree. (see Section 4 at page 19)
- A short paper submitted to an international conference. (see Section 3 at page 11)

We advise the reader to first read the short paper, and then to get a more complete information by reading the internship report.

1.4 Short paper : illustrative examples of the coverage approach

The short paper is aimed at explaining the principles of the approach and show its usefulness on 4 examples. The first three examples use the rewrite rules presented in section 2. They show that properties may often be satisfied by vacuity, i.e. by the absence of some of the events involved in the property. When the property can easily be satisfied by vacuity, it must be checked whether all traces exploit vacuity which may reveal a problem with the expression of the property.

The examples given in this short paper illustrate two cases where the vacuity revealed problems:

- the case of spelling errors in the name of events,
- a case where traces did not correspond to the system under review, and hence did not feature the expected events.

A third example discussed in this paper shows that the rewriting into disjunctive form may be applied to the original formula, but also to the negation of the formula. In such a case, it allows to distinguish between several cases which lead the property to fail. This classification of failures can be of interest to better understand failures.

The rewriting rules used for the first three examples are (close to) the ones described in the internship report (Section 4). The short paper goes further by defining new rewriting rules which focus on the number of occurrences of key events. A fourth example is given which uses these rules to get a finer decomposition of the formula. This leads to a better understanding of the set of traces which satisfy the formula and helps identify traces which satisfy unexpected sub-properties.

1.5 Internship report

The short paper does not list the rewriting rules. They can be found in the internship report. For ease of reading, we have also extracted these rules from the internship report and copied them into Section 2.

The internship report does not only list the rewriting rules, it provides additional contributions:

- related work on coverage techniques
- an informal presentation of PARTRAP
- a presentation of the coverage system based on a disjunctive form and the associated term rewriting system.
- the experimentation of the term rewriting system on several examples
- a preliminary proposal of translation from PARTRAP to first order logic that can be used by an SMT solver.

Experiments with the rewriting system showed that some properties could be rewritten into a disjunctive form where one or several disjuncts are always false. This leads to problems when a software tester tries to design a test suite such that its set of traces covers each of the disjuncts.

This motivated to link PARTRAP with an SMT solver. The goal of the solver is to find a trace such that a PARTRAP formula (which can be one of the disjuncts generated by the term rewriting system) can be verified by this trace. Preliminary work was carried out during this internship for a subset of PARTRAP. In the case of coverage, the SMT based approach helps figure out whether a disjunct is feasible or not.

1.6 Tooling

A prototype tool written in Haskell was implemented during the internship, and improved when writing the short paper. It covers the first level of decomposition, based on absence and occurrences, described in the short paper.

The gitlab repository of the Modmed project is at the following address:

`https://gricad-gitlab.univ-grenoble-alpes.fr/modmed`

The coverage prototype is available through the "coverage" branch of the Modmed interpreter project, on the gitlab repository. You can access it at the following address:

`https://gricad-gitlab.univ-grenoble-alpes.fr/modmed/partrap/tree/coverage`

The tool based on the SMT solver and a tool implementing the second level of decomposition are not available at this time.

References

[1] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. Coverage criteria for model-based testing using property patterns. In *MBT*, volume 141 of *EPTCS*, pages 29–43, 2014.

2 Revised rewriting system for ParTraP

The rewriting system given at chapter 4, section 4.3 of the internship report is duplicated here, with some corrections.

In order to rewrite the ParTraP properties into disjunctive form, we used a term rewriting system. This system will help decompose our properties while preserving the semantics of the language. The decomposition is based on classical rules and axioms applied by pushing the negations inwards, and distributing the disjunctions over the conjunctions.

We write rules as `A ↦ B`. It means the expression `A` is rewritten into its new form `B`. `B` must have the same semantics as A, in other words, if τ is a trace, $\tau \models A \Leftrightarrow \tau \models B$ where $\models$ is the satisfaction relation. The full set of rewrite rules is given below. These rules do not follow any specific order and thus make the system nondeterministic.

Following classical logic, double negation can be eliminated:

$$\texttt{not not } P \mapsto P$$

Applying de Morgan's laws, negations can be pushed inwards as in the following couple of rules.

$$\texttt{not } (P_1 \texttt{ or } P_2) \mapsto (\texttt{not } P_1) \texttt{ and } (\texttt{not } P_2)$$

$$\texttt{not } (P_1 \texttt{ and } P_2) \mapsto (\texttt{not } P_1) \texttt{ or } (\texttt{not } P_2)$$

The quantifiers `forall` and `exists` are complete duals, thus the negation of one is the other.

$$\texttt{not } (\texttt{forall } x \texttt{ in } e\texttt{, } P) \rightarrow \texttt{ exists } x \texttt{ in } e\texttt{, not } P$$

$$\texttt{not } (\texttt{exists } x \texttt{ in } e\texttt{,}P) \rightarrow \texttt{forall } x \texttt{ in } e\texttt{, not } P$$

Because `absence_of` is simply defined as the negation of `occurrence_of`, we can simplify them:

$$\texttt{not } (\texttt{occurrence_of } E\, x \texttt{ where } c) \mapsto \texttt{absence_of } E\, x \texttt{ where } c$$

$$\texttt{not } (\texttt{absence_of } E\, x \texttt{ where } c) \mapsto \texttt{occurrence_of } E\, x \texttt{ where } c$$

The last two constructs are `after` and `before` along with their strict versions. The same rewrite method can be applied to the `before` and its strict version, thus it will not be mentioned as well. `after` can be decomposed into two disjuncts:

$$\texttt{after o } E\,x \texttt{ where } c\texttt{, } P \mapsto \texttt{not (occurrence_of } E\,x \texttt{ where } c\texttt{) or}$$
$$\texttt{(after! o } E\,x \texttt{ where } c\texttt{, } P\texttt{)} \tag{1}$$

A property is said to be satisfied when using the `after` construct if it happens at anytime after the occurrence of the event at hand. Since we have a conjunction between two properties, P_1 and P_2, then both of them should be satisfied. Thus, the conjunction can be distributed here into two subexpressions:

$$\texttt{after o } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ and } P_2\texttt{)} \mapsto \texttt{(after o } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) and}$$
$$\texttt{(after o } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

Similarly, `after` can be distributed over a disjunction, but the rule only applies for `after first` and `after last` . It does not apply for `after each` because it would require the same property to hold after each event while the not distributed version requires only one of the properties to hold after each event.

$$\texttt{after first } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ or } P_2\texttt{)} \mapsto \texttt{(after first } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) or}$$
$$\texttt{(after first } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

$$\texttt{after last } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ or } P_2\texttt{)} \mapsto \texttt{(after last } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) or}$$
$$\texttt{(after last } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

Similar rules can be defined for the strict version of `after`

$$\texttt{after! o } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ and } P_2\texttt{)} \mapsto \texttt{(after! o } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) and}$$
$$\texttt{(after! o } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

$$\texttt{after! first } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ or } P_2\texttt{)} \mapsto \texttt{(after! first } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) or}$$
$$\texttt{(after! first } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

$$\texttt{after! last } E\,x \texttt{ where } c\texttt{, (}P_1 \texttt{ or } P_2\texttt{)} \mapsto \texttt{(after! last } E\,x \texttt{ where } c\texttt{, } P_1\texttt{) or}$$
$$\texttt{(after! last } E\,x \texttt{ where } c\texttt{, } P_2\texttt{)}$$

The next rule is the negation of the `after` construct:

$$\texttt{not (after o } E\, x \texttt{ where } c\texttt{, } P) \mapsto (\texttt{occurrence_of } E\, x \texttt{ where } c) \texttt{ and}$$
$$\texttt{not (after! o } E\, x \texttt{ where } c\texttt{, } P)$$

To prove this rule, we apply the above rewrite rules. First we apply rule 1 on the expression between the parenthesis we get:

$$\texttt{not (not (occurrence_of } E\, x \texttt{ where } c) \texttt{ or (after! o } E\, x \texttt{ where } c\texttt{, } P))$$

Then, pushing the negation inwards we end up with the two same disjuncts as in the rule.

The last rule presented is the negation of `after!` construct:

$$\texttt{not (after! first } E\, x \texttt{ where } c\texttt{, } P) \mapsto \texttt{after first } E\, x \texttt{ where } c\texttt{, not } P$$

$$\texttt{not (after! last } E\, x \texttt{ where } c\texttt{, } P) \mapsto \texttt{after last } E\, x \texttt{ where } c\texttt{, not } P$$

These rules are a little complicated to understand from the first look. Consider the property without the negation. It is satisfiable if an event occurs and the property holds on the subtrace covered. To negate this, either the event never happens or the property was not satisfiable in the subtrace covered. This is exactly the definition of the `after` construct.

Once again, it applies only to the `after! first` and `after! last` cases, because the negation of (`after! each` E, P) is satisfied as soon as one case does not satisfy the property (because `each` is a conjunction), while (`after each` E, `not` P) requires that all events satisfy (`not` P).

3 FM 2018 paper

This section includes a short paper submitted to an international conference:
Using coverage information to help understanding formal trace properties
Y. Blein, Y. Ledru, M. A. Tabikh, L. du Bousquet,R. Groz
under submission

Abstract ParTraP is a language to express and evaluate temporal properties on finite traces of parametric events. We propose a set of rewriting rules that allows expressing a ParTraP property into a disjunction of subproperties, which may be satisfied or not. We show that this coverage information can be of great help both in providing insights on a corpus of traces and a deeper understanding of temporal properties. Classical errors and unhandled corner cases are revealed quickly. We support this claim with a series of examples extracted from an industrial case study.

Using coverage information to help understanding formal trace properties

Y. Blein, Y. Ledru, M. A. Tabikh, L. du Bousquet, and R. Groz

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000, Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

Abstract. ParTraP is a language to express and evaluate temporal properties on finite traces of parametric events. We propose a set of rewriting rules that allows expressing a ParTraP property into a disjunction of sub-properties, which may be satisfied or not. We show that this coverage information can be of great help both in providing insights on a corpus of traces and a deeper understanding of temporal properties. Classical errors and unhandled corner cases are revealed quickly. We support this claim with a series of examples extracted from an industrial case study.

1 Introduction

Most computer systems can easily be augmented to produce traces of their activity. The verification of properties on these traces appears as a lightweight approach to formal methods [7]. Several languages based on temporal logic have been proposed to express trace properties such as FO-LTL, SALT or JavaMop [5,1,6]. Compared to those languages, ParTraP (Parametric Trace Property language) [2] is the only one that features the patterns of Dwyer et al. [4] and parametric events, i.e. events that carry information as parameters.

In this paper we propose a term rewriting system for ParTraP which performs the decomposition into a specific disjunctive normal form (DNF). Rewriting a property with this system allows providing some meaningful coverage feedback when the property is evaluated on a large set of traces. We propose two levels of decomposition and implement a prototype. The first level of transformation is similar to the one described by Li et al. for Linear Temporal Logic [8]. In this article, we successively present the two levels of transformation (section 2 and 3), and illustrate their benefits on an industrial case study of computer-aided surgery for total knee arthroplasty (TKA).

The TKA system is developed by the BlueOrtho company. It helps the surgeon to position cutting guides on the basis of trackers attached to the bones of the patient, and localized by a stereoscopic camera. TKA software has been augmented to log in a trace every interaction with the surgeon and significant events in the system activity. Until now, more than 10 000 surgery traces have been collected by BlueOrtho, mainly for post-market surveillance of their system. Each execution trace counts about 3000 events. In the following examples, several requirements have been formalised with ParTraP and evaluated on a sample of 100 TKA traces.

2 Decomposition Based on Absence and Occurrence

A property may be satisfied on trivial cases, known as vacuity. In Boolean logic this may occur for the implication $P \Rightarrow Q$ when P is false. Vacuous truth may induce a false sense of confidence in a system.

The ParTraP language is based on a nesting of temporal scopes and patterns. It is used to analyze traces produced in an industrial setting. We noticed that users tend to write complex properties with subtle cases, which may be intricate to detect. In particular, vacuous truth may be induced by the absence of certain events, and can easily be overlooked. For this reason, it was important to provide coverage information for the user to detect those cases.

On the same principle as $P \Rightarrow Q$ is equivalent to $\neg P \vee Q$, we defined a set of rewriting rules that allows transforming a ParTraP property into a disjunction of cases. Those cases distinguish between the absence or the occurence of key events in scopes. Our system, which includes more than 20 rules, is able to deal with all types of combinations of scopes and patterns. Instead of presenting the whole rule set, we chose to illustrate this principle and its benefits through three examples.

Example 1: Misspelled Identifier

In many formal languages, spelling mistakes in identifiers are detected by static checks. Unfortunately it is not the case with ParTraP. Since event type identifiers are not declared before being used in a property, misspelled identifiers cannot be detected statically. Let us consider a first example of requirement and its formalization as a trace property.

TKA relies on a set of uniquely identified trackers that can be localized thanks to the stereoscopic camera. The software governs their activation remotely. Each of them should only be activated if they have been properly detected in the past. This can be captured through the following ParTraP property[1]:

```
before each EnableTracker enable, occurrence_of
   TrackerDetected detect where detect.id == enable.id
```

In plain English, this property states that each event of type `EnableTracker` should be preceded by an event of type `TrackerDetected`, sharing the same `id` parameter. The identifier coming right after the type of an event allows naming the occurrences and accessing their parameters. The `where` clause ensures that events matching the preceding event type also respect the subsequent predicate. This property is satisfied by all the traces of the corpus. There are two ways for a trace to satisfy the property: either the trace does not feature any `EnableTracker` event and the property is vacuously true, or this event is preceded by a corresponding `TrackerDetected` event. Rewritten in DNF, the property forms a disjunction composed of two sub-properties:

[1] For space reasons, the semantics of ParTraP will be given informally through several examples. The interested reader may refer to the language specification [2].

(a) `absence_of EnableTracker enable`
(b) `before each! EnableTracker enable, occurrence_of TrackerDetected detect where detect.id == enable.id`

Sub-property (a) captures the potential absence of `EnableTracker` and (b) uses the stricter version `each!` that requires at least one occurrence of this event. Let us define the coverage of a sub-property as the number of traces in our sample that satisfy the sub-property. In this example, (b) is covered by 100 traces of our sample.

If `EnableTracker` was mistakenly written as `TrackerEnabled`, the new property would still be satisfied. This may lead to a false sense of confidence in the system. Evaluating the coverage of each sub-property would immediately reveal the mistake since only (a) would be covered.

Example 2: Aggregated Information on Failure

Whenever a property is violated on a given trace, the ParTraP interpreter describes as precisely as possible the facts that led to this verdict. Whereas this information may be useful when dealing with a few traces, it is clearly inapropriate to deal with a larger corpus such as the one gathered by BlueOrtho. On the contrary, evaluating the coverage of a property provides synthetic information over a set of traces. To understand which parts of a property caused a violation, one may evaluate the *coverage of the negation* of the property under study.

For instance, the precision of the stereoscopic camera of TKA is only guaranteed if its temperature stays within 20°C and 50°C. It can be expressed in ParTraP as follows:

```
absence_of CameraTemp t where t.val < 20 or t.val > 50
```

74 traces of our sample satisfy this property. Its negation consists in replacing `absence_of` with `occurrence_of` and can be rewritten in DNF to obtain a disjunction composed of two sub-properties:

(a) `occurrence_of CameraTemp t where t.val < 20`
(b) `occurrence_of CameraTemp t where t.val > 50`

Out of the 100 traces, sub-property (a) is covered by 10 while (b) is covered by 21. This means that 5 traces cover both sub-properties. This example shows that decomposing and studying the negation of the property brings interesting statistics on the actual system behavior.

Example 3: Heterogeneous Trace Corpus

When performing a surgery with the assistance of the TKA system, one critical step to a successful completion consists in acquiring the position of the hip center of the patient. This complex operation may be repeated by a surgeon until he is satisfied with the results. Thus some surgery traces contain several `HipCenter` events carrying the position of the point for each acquisition. For

technical reasons, there should not be any pair of hip center computations with resulting points that are too spread apart (here we arbitrarily chose 1.0 cm):

```
after each HipCenter h1, absence_of HipCenter h2 where
    dist(h1.point, h2.point) >= 1.0
```

The property is rewritten in DNF as the disjunction of the following two sub-properties, which distinguish the case where at least one event `HipCenter` occurs from the case where it does not:

(a) `absence_of HipCenter`
(b) `after each! HipCenter h1, absence_of HipCenter h2 where 1.0 <= dist(h1.point, h2.point)`

All the traces of the corpus but one satisfy the original property. The only exception results from misusage of the system by the surgeon. Coverage evaluation shows that (a) is covered by 6 traces while (b) is covered by 93 traces. The number of traces satisfying sub-property (a) is surprisingly high given the fact that hip center acquisition is critical to the completion of the surgery. Closer inspection of the 6 problematic traces revealed that they were actually traces generated by another product of the same manufacturer (dealing with shoulder surgery). It is only through coverage measurement that we noticed that the trace corpus we were given was heterogeneous.

3 Further Decomposition

In the previous section, we have shown that decomposing a property leads to better insights on its semantics. This allowed us to spot erroneous traces. The above decompositions attempt to exhibit a disjunctive form of the property by distinguishing between the presence or absence of some event in the trace. However, the presence of an event may correspond to one or more occurrences of this event. It is interesting to further decompose the property according to the number of occurrences of the events. This brings a finer decomposition by favoring the occurrence of more disjuncts in the property.

Let us consider a property of the form `after each` A`,` P_1 `or` P_2. It can be decomposed according to the number of occurrences of A. In particular, when considering the cases with 0, 1, 2 or more occurrences, we obtain a disjunction composed of the following sub-properties:

(a) `absence_of` A
(b) `occurrence_of {1}` A `and (after first` A`,` P_1 `or` P_2`)`
(c) `occurrence_of {2}` A `and (after first` A`,` P_1 `or` P_2`) and (after last` A`,` P_1 `or` P_2`)`
(d) `occurrence_of {3,}` A `and (after each` A`,` P_1 `or` P_2`)`

This disjunction can be further rewritten to exhibit several cases at the top level:

(a) `absence_of` A

(b) `occurrence_of {1}` A `and (after first` A, P_1`)`
(c) `occurrence_of {1}` A `and (after first` A, P_2`)`
(d) `occurrence_of {2}` A `and (after first` A, P_1`) and (after last` A, P_1`)`
(e) `occurrence_of {2}` A `and (after first` A, P_1`) and (after last` A, P_2`)`
(f) `occurrence_of {2}` A `and (after first` A, P_2`) and (after last` A, P_1`)`
(g) `occurrence_of {2}` A `and (after first` A, P_2`) and (after last` A, P_2`)`
(h) `occurrence_of {3,}` A `and (after each` A, P_1 `or` P_2`)`

To illustrate the usefulness of this decomposition, let us consider a final example from the case study. In TKA, the `types` parameter of the `SearchTracker` event lists the required types of trackers to perform the surgery. Each of these trackers types should be detected at least once before starting the acquisitions. An attempt to formalize this requirement could result in the following ParTraP property:

```
after each SearchTrackers st,
before first StartAcquisition,
forall ty in st.types,
occurrence_of TrackerDetected td where td.ty == ty
```

As mentioned earlier, in the `before first` E clause we can distinguish the cases whether E occurs or not. Thus we can rewrite the previous property as

`after each SearchTrackers st, (absence_of StartAcquisition or before first! StartAcquisition,` P`)`, where

$P =$ `forall ty in st.types,`
`occurrence_of TrackerDetected td where td.ty == ty`.

It is now precisely of the form `after each` A, P_1 `or` P_2 and can be decomposed in the same manner. For space reasons, we only present some of the sub-properties (the others can be derived easily):

(b) `occurrence_of {1} SearchTrackers and (after first SearchTrackers st, absence_of StartAcquisition)`
(c) `occurrence_of {1} SearchTrackers and (after first SearchTrackers st, before first! StartAcquisition,` P`)`
(f) `occurrence_of {2} SearchTrackers and (after first SearchTrackers st, before first! StartAcquisition,` P`) and (after last SearchTrackers st, absence_of StartAcquisition)`

Sub-property (c) encodes the nominal case: a single set of trackers is looked for and found at once before starting any acquisition. Covered by 55 traces, it is also the most common case in our sample. Both (b) and (f), each covered by 2 traces, look more suspicious. Indeed, sub-property (b) says that the acquisition phase never started after looking for a set of trackers. Closer inspection reveals that the two traces satisfying this case were actually generated by the shoulder product, mentioned in the previous section, and use a different name for event `StartAcquisition`. Sub-property (d), not given here, is similar and also exhibits

traces from the shoulder surgery. Finally, sub-property (f) says that an event `StartAcquisition` is found once but is missing after the second search for new trackers. The two traces satisfying this surprising case revealed that we overlooked a special case when writing the property: in a recent version of the TKA product, event `StartAcquisition` takes a different name if we reconnect the trackers in mid-surgery. This example shows that a fine decomposition of the property helps to better understand the variety of traces.

4 Conclusion

ParTraP attemps to ease temporal specification by featuring high level patterns and a verbose syntax. Nonetheless, expressing temporal properties remains a difficult problem and cannot be solved by a language alone. Some properties that may seem easy to understand can divert the reader from careful consideration of special cases.

In this work, we proposed to combine the decomposition of a property in disjunctive normal form with coverage information. We identified two levels of decomposition. The first one splits a property into a disjunction of sub-properties according to the presence or absence of key events in temporal relations (scope). In particular, it allows distinguishing vacuously true situations from the main constraint. The second level of decomposition goes further by splitting the property according to the number of occurrences of those same key events, and provides a finer grain understanding of the property. The term rewriting system has been implemented on top of the ParTraP interpreter[2]. We demonstrated the usefulness of the approach through several examples extracted from an industrial case study: it helps to better understand temporal properties and the results of their evaluation on a corpus of traces, and to identify faulty or incomplete properties.

The idea of measuring coverage of temporal properties based on Dwyer's patterns has been explored by Cabrera Castillos et al. [3]. They transform temporal properties into automata and measure the coverage of these automata by a test suite. Instead we proposed a rewriting system which keeps the expression of properties in the original ParTraP language. Hence the user does not need to master a different formalism to understand coverage information.

As future work, we intend to complement this coverage approach with techniques of example and counter-example generation to provide different viewpoints on the understanding of temporal properties.

Acknowledgments

This work is funded by the ANR MODMED project (ANR-15-CE25-0010).

[2] https://gricad-gitlab.univ-grenoble-alpes.fr/modmed/partrap

References

1. Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT - structured assertion language for temporal logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.
2. Yoann Blein, Yves Ledru, Lydie du Bousquet, Roland Groz, Arnaud Clère, and Fabrice Bertrand. MODMED WP1/D1: Preliminary Definition of a Domain Specific Specification Language. Technical report, LIG, MinMaxMedical, BlueOrtho, 2017.
3. Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. Coverage criteria for model-based testing using property patterns. In *MBT*, volume 141 of *EPTCS*, pages 29–43, 2014.
4. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999.
5. Sylvain Hallé and Roger Villemaire. Runtime monitoring of message-based workflows with data. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 63–72. IEEE Computer Society, 2008.
6. Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. Javamop: Efficient parametric runtime monitoring framework. In *ICSE*, pages 1427–1430. IEEE Computer Society, 2012.
7. Cliff B. Jones. Formal methods light. *ACM Comput. Surv.*, 28(4):121, 1996.
8. Jianwen Li, Geguang Pu, Lijun Zhang, Zheng Wang, Jifeng He, and Kim Guldstrand Larsen. On the relationship between LTL normal forms and büchi automata. In *Theories of Programming and Formal Methods*, volume 8051 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2013.

4 Internship report of Mohammad Ali Tabik

Measuring the Coverage of Trace Properties
M. A. Tabikh
Internship report for the Master of Science in Informatics at Grenoble
June 2017

Abstract The verification of Medical Cyber Physical Systems (MCPS) is essential to proving their safety. Formal methods can be very useful in the process. However, it is very costly and almost impossible to be used with systems functioning in variable conditions. For this purpose, a new language ParTraP was introduced to monitor the traces properties generated by MCPS. This report is dedicated to define a coverage metric for these trace properties by introducing a term rewriting system. The validity of this system is illustrated through a detailed example. A preliminary work using SMT solvers is then introduced to improve the results. This work is in an advanced stage and is promising.

Master of Science in Informatics at Grenoble
Master Informatique, Université Grenoble Alpes
Specialization Advanced Information Systems and Software Engineering

Measuring the Coverage of Trace Properties

Mohammad Ali Tabikh

June 21, 2017

Masters research project performed at
VASCO/LIG

Under the supervision of:
Yves Ledru - LIG

Defended before a jury composed of:
Sophie Dupuy-Chessa
Ioannis Parissis
Hubert Garavel
Marius Bozga

June 2017

Abstract

The verification of Medical Cyber Physical Systems (MCPS) is essential to proving its safety. Formal can be very usefull in the process, however, it is very costly and almost impossible to be used with systems functioning in variable conditions. For this purpose, a new language ParTraP was introduced to monitor the traces properties generated by MCPS. This report is dedicated to define a coverage metric for these trace properties by introducing a term rewriting system. The validity of this system is illustrated through a detailed example. A preliminary work using SMT solvers is then introduced to improve the results. This work is in an advanced stage and is promising.

Résumé

La vérification de Systèmes Cyber-Physiques Médicaux (SCPM) est essentielle pour prouver leur sûreté. L'utilisation de méthodes formelles peut contribuer à ce processus, mais elles sont difficiles à mettre en oeuvre dans ces systèmes dont les conditions d'utilisation peuvent varier fortement. C'est pourquoi le projet MODMED a proposé un nouveau langage, ParTraP pour spécifier les propriétés des traces générées par les SCPM. Ce rapport définit une mesure de couverture des propriétés d'une trace, mise en oeuvre par un système de réécriture. Ce système a été expérimenté sur un exemple détaillé. Ce rapport présente également une amélioration de cette mesure de couverture en utilisant un solveur SMT. Cette amélioration est bien avancée et prometteuse.

Acknowledgement

I wish to express my sincere thanks to my supervisor Yves Ledru for the direction of my master thesis and for the many things he taught me. Benefiting from his experience and his help was very valuable. I am also grateful to Yoann Blein for his contributions and follow up during this work.
Finally, I thank all the VASCO team for welcoming me as a member for the past 5 months.

Contents

Abstract **i**

Résumé **i**

Acknowledgement **ii**

1 Introduction **1**
1.1 The MODMED Project 2
1.1.1 Goals 2
1.1.2 Approach 2
1.1.3 Case Study 3
1.2 Coverage Evaluation for ParTraP 3
1.3 Report Structure 4

2 Code Coverage **7**
2.1 Coverage Criteria 7
2.1.1 Function Coverage 7
2.1.2 Loop Coverage 8
2.1.3 Statement Coverage 8
2.1.4 Decision Coverage 8
2.1.5 Condition Coverage 8
2.1.6 Multiple Condition Coverage 9
2.1.7 Condition / Decision Coverage 9
2.1.8 MCDC: Modified Condition/Decision Coverage 9
2.1.9 Path Coverage 10
2.1.10 Linear Code Sequence and Jump (LCSAJ) 11
2.1.11 Synchronization Coverage 11
2.2 Comparison of Coverage Criteria 11
2.3 Linear Temporal Logic 13
2.3.1 Unique First Cause Coverage 13
2.3.2 LTL Coverage Using State Machines 14
2.4 DNF: Disjunctive Normal Form 14

3 Domain Specific Language - ParTraP **15**
3.1 Introduction 15
3.2 Syntax 15
3.3 Informal semantics 16
3.3.1 Events 16
3.3.2 Patterns 17
3.3.3 Scopes 17
3.3.4 Timed Variants 17
3.3.5 Quantifiers 18
3.3.6 Event Selection 18
3.4 Semantics 18

4 A Disjunctive Based Coverage System for ParTraP **19**
4.1 Using Disjunctive Normal Form - DNF 20
4.2 Language Modifications 20
4.3 Term Rewriting System 21
4.4 Example 23

5 Implementation and Case Study **25**
5.1 Implementation Procedure and Discussion 25
5.1.1 Discussion 28
5.2 Side Effects of the System 29
5.2.1 Example 1 29
5.2.2 Example 2 29

6 Working with SMT, Preliminary Work **31**
6.1 Introduction SMT Solvers 31
6.2 Translation From ParTraP to First Order Logic 32
6.3 Example and Discussion 34

7 Future Work and Conclusion **37**
7.1 Conclusion 37
7.2 Future Work 37

Bibliography **39**

A Appendix **43**
A.1 ParTraP Semantics 43
A.1.1 Preliminary Definitions 43
A.1.2 Semantics Rules 44
A.2 Modified ParTraP Grammar 46
A.3 Modified ParTraP Semantic Rules 47

— 1 —
Introduction

Medical Cyber-Physical Systems (MCPS) are powerful solutions used for improving healthcare services supporting complex medical interventions. They combine data gathered from novel sensors and existing modalities like scanners with elaborate software processing to assist the users in their work. They are used in the same manner as a Flight Management System helps a pilot fly an aircraft. One example is Blue Ortho's MCPS, it aids the surgeons in performing Total Knee Arthroplasty (TKA) more precisely, increasing the surgery's success rates and accuracy and potentially dividing the number of revisions by two. This is extremely beneficial for the patients, because a second surgery replacing the first prosthesis might lead to significant bone damage and prevents them from walking normally for the rest of their lives.

Such system complexity requires rigorous testing and continuous follow up and validation of its correctness. However, the safety of such MCPS is hindered by the current software verification practices by the industry. The Pacemaker Challenge [MSW14] recently demonstrated that it is theoretically, if not economically, possible to perform full verification of some medical devices which interact in sufficiently restricted and controlled ways with their environment. Yet, such systems, as the one we are studying, work in varied conditions where full formal verification within their environment becomes too costly or even impossible. Two main differences between MCPS and other safety-critical systems are:

- Pilots are trained to fly specific planes using predefined procedures, however, a surgeon might combine several medical devices and various procedures to perform his work, thus it would be difficult to completely verify the whole system a priori.

- Som regulatory administrations require evidences of safety a priori, like the FDA. On the other hand, current MCPS regulations do not require such rigorous proofs.

One unique opportunity MCPS provide is the verification process, especially in the clinical context of their lifetime. This can be done by exploiting their execution traces to provide us with an unbiased and precise understanding of their behavior in the field. The inforrmation in this chapter are inspired from the MODMED project proposal [Gen15].

1.1 The MODMED Project

MODMED is a collaborative project between the public and private sector. It was initiated by French small sized enterprises working in the medical devices industry for the purpose of creating high-confidence medical devices. MODMED aims at adapting formal methods and tools to verify their safety properties more efficiently and reliably. In addition to Laboratoire d'Informatique de Grenoble (LIG), there are two partners, MinMaxMedical (MMM) and Blue Ortho (BO).

1.1.1 Goals

MODMED is an industrial research, that aims to acquire applied knowledge and new skills by:

- Helping small enterprises in applying these techniques in their software verification for a lower cost.
- Applying these methods and tools to Blue Ortho MCPS.
- Advertising the results to the MCPS industry and software engineering community.
- Increasing the safety measures taken in the development of MCPS.

In addition, this project possesses numerous expected benefits for MCPS [Mod15]:

- Critical pre-requisites and requirements will be best understood by the formalisation of trace properties.
- System tests will be more efficient when supervised by trace properties expressing the pre-requisites and the oracle of each test.
- Post-market surveillance will be more relevant than using user surveys.
- Properties will allow classifying real medical interventions traces and quantifying MCPS usage.
- Troubleshooting recurring problems will be automated.

1.1.2 Approach

The ANR MODMED initiative was created to improve the instrumentation of such systems, and to create execution traces to monitor them. This projects embraces "lightweight formal methods" to bridge the gap between MCPS industry and safety-critical systems that already apply formal methods. It focuses on methods and tools allowing the verification of a selection of functional and safety requirements identified as most critical by quality engineers: partial formal modeling, continuous monitoring, test assessment and trace analysis.

At this stage, the project has identified 15 properties which are representative of the application domain. It has designed a language, based on patterns and temporal logic, able to express most of these properties. A prototype implementation of the language is under way.

The new language, named ParTraP, and its toolset were developed to express the correctness properties exhibited in these. It is directly applicable in medical systems, as the partners have already collected thousands of traces of their system that is used in knee surgeries.

1.1.3 Case Study

This project focuses on Blue Ortho's Total Knee Arthroplasty (TKA) MCPS. It was involved in the MODULHIPS ANR 2012 project that developed a novel process for Total Hip Prosthesis operations. The surgery involves replacing parts of the knee joint with a prosthesis. Its purpose is to relieve the pain of an arthritic knee, while preserving its functionality. This helps the surgeon achieve precise cuts through a guide for the installation of cutting guides. The system establishes a spatial reconstruction, combining it with the desired objective of the surgeon to determine the exact cutting position. It is currently deployed in many countries and have been used in thousands of surgeries.

The system is composed of: touch screen connected directly to the machine, a 3D camera, a set of trackers and a set of mechanical instruments for attaching trackers and cutting guides to the tibia and the femur. There are several steps that are carried out by the surgeon in each surgery. These steps are configurable into a $profile$ defining the surgeon's operating preferences. However, every surgery should contain the following steps: sensor calibration, acquisition of anatomical points, checking acquisitions, adjustment of target parameters, and cutting guides setting.

The traces collected from the executed surgeries will be analyzed in order to understand the requirements and check that they can actually be verified using execution traces. This will help Blue Ortho build their new MCPS which is similar to the TKA one.

1.2 Coverage Evaluation for ParTraP

Properties expressed in the language can be evaluated but it remained to be defined a coverage measurement for them, in other terms, how much of a property is covered. This is done by performing an analysis of existing traces and checking whether they fulfill properties identified during the specification or the design of the system. This work is dedicated to this aspect.

The goal of the internship is to define coverage measurements for the trace properties. Given a trace, or a set of traces, and a property, what is the coverage of the property by the given traces, *i.e.* how much of the property was exercised by the traces. This requires to define a coverage criterion for the properties and the tools in order to determine the exact cause of satisfiability. Several directions can be explored to define the coverage metrics: decomposition of properties, measurement of coverage of state machines, measurement of coverage of code implementing the properties.

[1]https://www.exac.com/products/knee/advanced-surgical-instrumentation

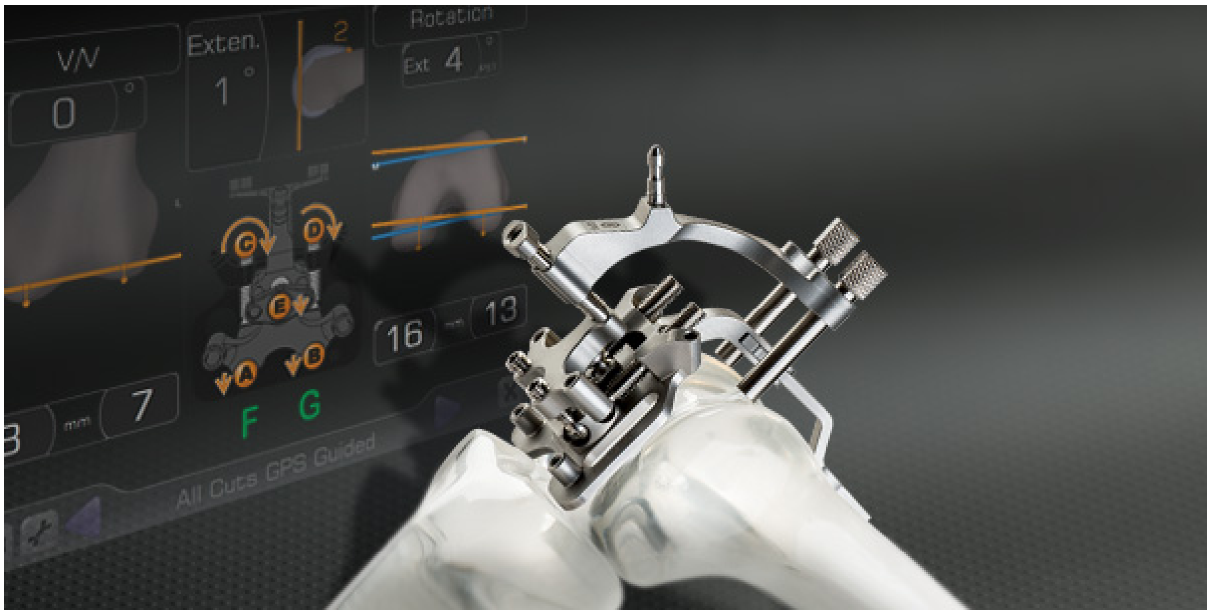

Figure 1.1: Total Knee Arthroplasty[1]

This paper will be presenting the decomposition of properties into a set of disjunct subproperties. We will be defining a term rewrite system that should preserve the semantics of our language. The resulting disjuncts will then be tested against a set of trace files to calculate their coverage. Followed by a short discussion about the results obtained. Then, it will introduce preliminary work done to remove non necessary disjuncts, thus improving the quality of the output.

1.3 Report Structure

This report is organized as follows:

Chapters 2 and 3 present the background of this work. More specifically, Chapter 2 details the state of the art in available coverage metrics. It states the most commonly used ones and some of their advantages and disadvantages along with some examples of their usage. Chapter 3 presents the newly developed domain specific language specifically for the MODMED project. It presents the basis it was built upon, its syntax and semantics.

In Chapter 4, we propose a method for measuring the coverage of the properties expressed in the created DSL. It introduces the usage of disjunctive based coverage for this purpose, in addition to modifications made to the language. Then, it presents all rewriting rules created to break down the properties into a set of disjunctive terms and expressions. Finally, we give an example of the implementation of such rules, along with a small discussion about the output generated.

In Chapter 5, we present the implementation procedure followed in proving the strength and validity of our rewriting rules. In addition, we discuss the results obtained

on a small example, while mentioning some issues discovered such as redundancy within the output.

In Chapter 6, we present a preliminary work on the improvement of our implementation. We introduce the SMT solvers, which are used in system verification. Then, we present a set of rewrite rules created as a middle step between creating the disjunctive expressions and testing the satisfiable ones over a set of test traces. Additionally, we discuss its usefulness and future possibilities.

Finally, Chapter 7 introduces some directions to explore and concludes this work.

— 2 —
Code Coverage

In order to ensure high quality of software, the developers have to apply various techniques throughout software development life-cycle. Checking the quality and correctness of them is a must to ensure minimal to null failures in execution. Code coverage is a mean to evaluate the completeness of a test suite.

This is done by executing pieces of source code over a predefined set of test cases to identify code segments which are not exercised by existing tests and therefore may prompt creation of additional test case [Mye79]. In addition, there is requirements-based coverage, where in any given test suite, there must exist at least one test per requirement that passes. For example, "The red light is turned on when the camera is recording". A test validating the requirement might follow this scenario: (1) Turn the camera on, and (2) Verify the red light is turned on. Another simple, yet unhelpful, test verifying the requirement is just to keep the camera off. Another method is functional testing which focuses on the overall program accomplishments with regard to the requirements proposed, however, it shall not be discussed in this paper.

In the following sections, we introduce some the existing code coverage criteria, and compare some of them to see their common usages. In addition, we introduce some of their advantages and disadvantages.

2.1 Coverage Criteria

There exist many coverage criteria, however, we will be discussing a few of them below (function, loop, statement, decision, condition, multiple condition, condition/decision and modified condition/decision), in addition to Unique First Cause coverage. They are very well known and have been used for a long time.

2.1.1 Function Coverage

Function coverage is mainly used to measure the number of functions covered called/assessed during the execution of a test suite, divided by the number of overall functions in a program. There are several variations of this criterion. One is BullseyeCoverage, which

considers a function covered if it was entered at least once [Cor11]. Another is gcov, this measures the total number of times a function was entered and exited. It is important to note that function coverage is not the same as functional coverage.

2.1.2 Loop Coverage

Loops are a major construct of several languages. They allow the programmer to iterate over a set of elements [KP13]. An interesting coverage criterion is to check if the loop body has been skipped, or else how many times it was executed. Thus, it checks the loop boundaries have been properly tested.

2.1.3 Statement Coverage

This method makes sure that every statement of a program is invoked at least once during testing. It is also known as line coverage, segment coverage [Nta88b] and C1 [Bei90]. It is a weak coverage criterion which can be easily satisfied in some cases, however certain behaviours will not.

```
num = -1;
if (condition)
    num = 5;
return num;
```

Figure 2.1: Example of statement coverage

The code in Figure 2.1 is satisfiable with only one test case (the condition is set to true) achieving 100% statement coverage. However, the behaviour where the condition is false is not tested.

2.1.4 Decision Coverage

This criterion reports whether every decision is evaluated to true and false, improving the output with respect to statement coverage. These decisions are in the form of control structures (such as the if-statement and while-statement). It is also known as branch coverage, all-edges coverage [Rop94] and decision-decision-path testing [Rop94]. A disadvantage of this criterion is it ignores branches within Boolean expressions, especially ones containing logical-or operators.

The code in Figure 2.2 is satisfiable with two test cases (A = true, B = false) and (A = false, B = false). However, the effect of B is never tested, which is equivalent to testing only A.

2.1.5 Condition Coverage

This criterion reports whether every boolean subexpression is evaluated to true and false. However, this does not ensure decision coverage.

```
if A or B then
    true_statement;
else
    false_statement;
endif;
```

Figure 2.2: Example of decision coverage

```
if A and B then
    true_statement;
else
    false_statement;
endif;
```

Figure 2.3: Example of condition coverage

The code in Figure 2.3 is satisfiable with two test cases (A = true, B = false) and (A = false, B = true). However, these cases do not satisfy decision coverage where the true statement is never evaluated.

2.1.6 Multiple Condition Coverage

This criterion requires test cases that include all combinations of inputs to a decision to be executed at least once, requiring exhaustive testing of the input combinations of a decision. It is one of the best structural coverage measures as it covers all possibilities, but it is impractical for a decision with a high number of inputs.

For example, the code in Figure 2.3 is satisfiable with four test cases (A = true, B = false), (A = false, B = true), (A = true, B = true) and (A = false, B = false). As you can see, n inputs require 2^n tests.

2.1.7 Condition / Decision Coverage

This criterion is a mixture of both Condition and Decision Coverage criteria. Satisfying means having test cases covering both criteria. For example, the test cases (A = true, B = true) and (A = false, B = false) in Figure 2.3 meet the coverage criterion. However, this does not allow the tester to distinguish which was the correct expression. For example, it may be A, B,(A or B) or (A and B).

2.1.8 MCDC: Modified Condition/Decision Coverage

In addition to the requirements of Condition/Decision coverage, Modified Condition/Decision coverage requires that each condition should be evaluated, at least twice, affecting

the decision's outcome independently relative to other conditions. This metric was created in conformance with the international technical standard DO-178B [RB11], which requires a full 100% coverage to award the certification.

Thus to have MCDC four conditions need to be met [Chi94] [KJDSJJLK01]:

- Every point of entry and exit in the program has been invoked at least once
- Every condition in a decision has taken all possible outcomes at least once
- Every decision in the program has taken all possible outcomes at least once
- Each condition in a decision has been shown to independently affect that decision's outcome

A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. [Chi94]

A very important advantage of this criterion over Multiple Condition coverage is it requires much fewer test cases. For n conditions, a minimum of $n+1$ test cases is needed. Going back to Figure 2.3, three test cases where (A = true, B = true), (A = true, B = false) and (A = false, B = true) provide MCDC.

One drawback of this criterion is that a condition may occur more than once in a decision. For example, ((not A and B) or A) where "A" and "not A" are coupled. This contradicts with the rule of the independent effect of each condition varied, this version is called Unique-Cause.

Another version of MCDC was introduced relaxing its requirements called Masking MCDC, it allows more than one input to change in an independence pair, as long as the condition of interest is shown to be the only condition that affects the value of the decision outcome.[Tea01]

2.1.9 Path Coverage

A path is a unique sequence of branches from the function entry to the exit [Cor11]. This criterion checks all pathes in each function and reports whether they have been followed. It is also known as predicate coverage.

A problem with loops is that they may introduce an infinite number of paths. This criterion avoids this problem by considering only a limited number of them. There are many variations of this criterion, one is Boundary-interior path testing which considers two possibilities for loops: zero repetitions and more than zero repetitions [Nta88a].

Although it requires very thorough testing, which is advantagous, it has two disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 12 if-statements has 4096 paths to test. Each new if-statement added would double the number of paths. The second disadvantage is that many paths are impossible to exercise due to relationships of data [Cor11]. To deal with

the large number of paths, many variations have been created of this criterion. Two important variations are linear code sequence and jump (LCSAJ) coverage and data flow coverage.

2.1.10 Linear Code Sequence and Jump (LCSAJ)

This criterion is a variation of path coverage that considers only sub-paths that can easily be represented in the program source code, without requiring a flow graph. LCSAJ executes source code lines in sequence, where they may contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs [Cor11]. This metrics main advantage is that it lacks the exponential explosion of paths in path coverage. However, this does not prevent infeasible paths.

2.1.11 Synchronization Coverage

This criterion measures if at each contention point, such as lock acquisition, waiting on a semaphore or in a monitor, there were both non-blocked and blocked threads. The idea behind this criterion is to ensure that source code has been verified in concurrent environment.[KP13]

2.2 Comparison of Coverage Criteria

The above criteria are implied or subsumed within others. For example, decision coverage implies statement coverage in case no unconditional jumps, such as goto statements or try-catch blocks are used. Another is statement coverage that implies decision coverage in case all if-then statements have an else branch, and there are no empty branches [KP13]. Figure 2.4 presents the ordering of control-flow code coverage criteria.

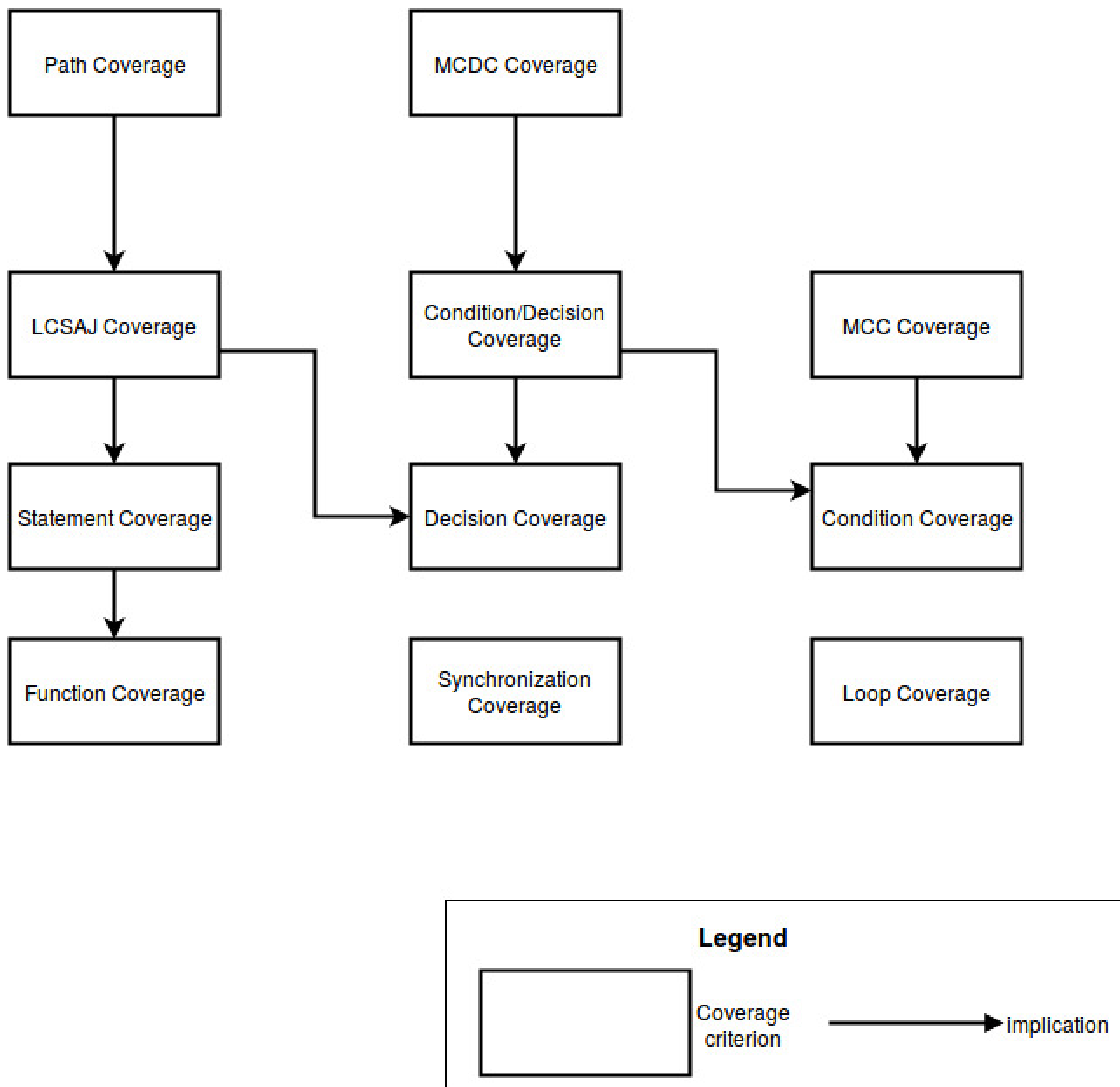

Figure 2.4: Relationship between the presented coverage criteria.

2.3 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu77] is a specification language dedicated to describe the properties of systems. Contrary to Boolean logic that expresses static properties, LTL is used to express dynampic properties. Some operators used in LTL are: G (always) denoted by $\Box$, F (eventually) denotedby $\Diamond$, X (next) denoted by $\bigcirc$, U (until) and R (release). LTL is defined by the grammar [LPZ+13]:

$$\varphi ::= \; a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi U \varphi \mid \varphi R \varphi \mid X\varphi$$

Where a belongs to the set of atomic properties AP, and φ is an LTL formula. Let $u = u_0u_1u_2...$ be a word in the infinite sequence $\xi \in \Sigma^\omega$ with $\Sigma = 2^{AP}$. The semantics of LTL is given below [GO01]:

- $u \models a$ `if` $a \in u_0$
- $u \models \neg\varphi$ `if` $u \not\models \varphi$
- $u \models \varphi_1 \wedge \varphi_2$ `if` $u \models \varphi_1$ `and` $u \models \varphi_2$
- $u \models \varphi_1 \vee \varphi_2$ `if` $u \models \varphi_1$ `or` $u \models \varphi_2$
- $u \models \varphi_1$ `U` φ_2 `if` $\exists i \geq 0, u_iu_{i+1}... \models \varphi_2$ `and` $\forall 0 \leq j < i, u_ju_{j+1}... \models \varphi_1$
- $u \models X\varphi$ `if` $u_1u_2... \models \varphi$

The release (R) operator is defined by $\varphi_1 R \varphi_2 = \neg(\neg\varphi_1 U \neg\varphi_2)$. If φ is neither a disjunction nor a conjunction then it is called a temporal formula.

2.3.1 Unique First Cause Coverage

This criterion has adapted masked MCDC to temporal logic properties by Whalen *et al.* [WRHM06], to rigorously cover the effect of each atomic condition in complex requirements. It measures the number of test cases in a test suite satisfiable by the given property. To be precise, if a property can be satisfied in more than one way, only the first cause would be sufficient for a test case to pass. To further define it, consider a path π satisfied by a formula A, a clause x is the unique first cause of A, if, in the first state along with π, x satisfies A [FW09].

For example, let's consider the property $\Diamond(x \vee y)$, *i.e.* there exists a state in the sequence that eventually satisfies the expression. Expressing this property on the sequence $\langle(x, \neg y), (\neg x, \neg y), (\neg x, y), (x, y)\rangle$, may have three states satisfying it by having either (x), (y) or $(x$ and $y)$ true. However, it suffices to say that the first state, $(x, \neg y)$, satisfies the property and thus x is the unique first cause.

According to [WRHM06], UFC coverage over a test suite is achieved if executing the test cases will guarantee that every basic condition in a formula has taken on all possible outcomes at least once, and each basic condition has been shown to independently affect the formula's outcome.

2.3.2 LTL Coverage Using State Machines

Translating LTL properties into state machines is a well known method for model checking of linear temporal requirements. The process starts by decomposing the properties first then creating the state machines. A very well known research done for this process is detailed in [GO01] and [BKRS12] and will not be detailed here. There are several algorithms following the translation, to cover the resulting automata. For the sake of this work, we prefer a simpler and faster solution. For this, we were interested in the first step of the translation, specifically, the translation of LTL into a set of disjunctive terms (DNF)[DP90].

2.4 DNF: Disjunctive Normal Form

Disjunctive Normal Form (DNF) is the standardization of a logical formula in Boolean mathematics [DP90]. It is widely used in areas such as automated theorem proving. For a logical formula to be in disjunctive normal form, there must exist a disjunction of one or more conjunctions of one or more literals. In addition, a DNF is said to be in full disjunctive normal form if each variable appears once per clause. The propositional operators in disjunctive normal form are: AND, OR and NOT.

The reduction of any Boolean term to DNF relies on applying the laws of Boolean algebra. According to B. A. Davey, H. A. Priestley [DP90], there exist several usable steps for such reduction, in addition to double negative elimination:

- Use de Morgan's laws to reduce a term to literals combined by joins and meets
- Use the distributive laws repeatedly, with the lattice identities, to obtain a join of meets of literals
- Drop any terms containing both x_i and its negation, for any i in the Boolean term $p(x_1, ..., x_n)$

One might consider a downside of DNF is the potential exponential explosion of the formula. However, for the sake of our work, it was very important to generate all possible expressions that would satisfy the original term. Consider the boolean expression below:

$$\begin{aligned}(A \vee B) \wedge (C \vee D) \wedge (E \vee F) \equiv \\ & (A \wedge C \wedge E) \ \vee \ (A \wedge C \wedge F) \ \vee \\ & (A \wedge D \wedge E) \ \vee \ (A \wedge D \wedge F) \ \vee \\ & (B \wedge C \wedge E) \ \vee \ (B \wedge C \wedge F) \ \vee \\ & (B \wedge D \wedge E) \ \vee \ (B \wedge D \wedge F)\end{aligned}$$

The expression was transformed into DNF. The explosion of the formula is obvious here, where the original expresseion was composed of the three conjunctive subexpressions, and ended up with eight disjunctive subexpressions.

— 3 —
Domain Specific Language - ParTraP

3.1 Introduction

The MODMED project requires a language to make formal requirements writable by software engineers with no training in formal methods and readable by domain experts. For this purpose, our team has developed a high-level language dedicated to property specification for MCPS. We call this new DSL: ParTraP (Parametric Trace Property language) designed to fulfill the requirements of the ExactechGPS-TKA case study, which is further detailed in [BBdB+16]. The information in this chapter is acquired from [Ble17].

Our approach was mostly influenced by the work of Dwyer *et al.* on *specification patterns* [DAC99]. They identified a set of frequently used patterns in real-world specifications and elaborated a pattern system, much like design patterns in software engineering. Basically, a pattern is a *requirement* (*e.g.*, an event triggers another one) coupled with a *temporal scope* (*e.g.*, after a certain event).

Dwyer *et al.* defined an LTL, linear temporal logic (LTL) [Pnu77], translation for each and every couple requirement/scope. The pattern system enables inexperienced users to write formal specifications in a natural language. However, it suffers from a limited expressiveness mostly due to the fact that scopes cannot be nested. Extending the system with a new requirement or scope requires writing the LTL translation for every possible couple containing the new construct, which is inconvenient and error-prone. Taha *et al.* shown that even the original pattern translations are inconsistent [TJD+15].

The proposed DSL is an event-based formalism that has a well-defined syntax, which is verbose in order to ease its understanding by software engineers not trained in formal methods. Mostly influenced by Dwyer et al [DAC99], we extended their expressiveness significantly by allowing them to be combined and nested, and to operate on parametrized events.

3.2 Syntax

The syntax of the proposed DSL is described by the grammar in figure 3.1.

```
⟨prop⟩      ::= ⟨pattern⟩
              | ⟨scope⟩ ‘,’ ⟨prop⟩
              | (‘forall’ | ‘exists‘) <ident> ‘in’ ⟨expr⟩ ‘,’ ⟨prop⟩
              | ‘(’ ⟨prop⟩ ‘)’
              | ‘not’ ⟨prop⟩
              | ⟨prop⟩ (‘and’ | ‘or’ | ‘equiv’ | ‘implies’) ⟨prop⟩

⟨scope⟩     ::= [‘within’ ⟨duration⟩] (‘after’ | ‘before’) (‘each’ | ‘first’ | ‘last’)
                ⟨event⟩
              | ‘between’ ⟨event⟩ ‘and’ ⟨event⟩
              | ‘since’ ⟨event⟩ ‘until’ ⟨event⟩

⟨pattern⟩   ::= ‘absence_of’ ⟨event⟩
              | ‘occurrence_of’ ⟨expr⟩ ⟨event⟩
              | ⟨event⟩ ‘followed_by’ ⟨event⟩ [‘within’ ⟨duration⟩]
              | ⟨event⟩ ‘precedes’ ⟨event⟩ [‘within’ ⟨duration⟩]
              | ⟨event⟩ ‘prevents’ ⟨event⟩ [‘for’ ⟨duration⟩]

⟨event⟩     ::= ⟨ident⟩ [⟨ident⟩ [‘where’ ⟨expr⟩]]
              | ‘set’ ‘(’ ⟨ident⟩ [⟨ident⟩] (‘,’ ⟨ident⟩ [⟨ident⟩])* ‘)’ [‘where’ ⟨expr⟩]

⟨duration⟩  ::= ⟨expr⟩ (‘ms’ | ‘s’ | ‘min’ | ‘h’ | ‘d’)
```

Figure 3.1: Syntax of the proposed DSL [Ble17]

The following expressions are typical examples that can be derived from this grammar:

- `after each A, B followed_by C`
- `after first A a where a.x != 0, absence_of B or absence_of C`
- `before last A a, forall v in a.set, occurrence_of 2 B where b.p == v`
- `between A a and B b where a.v == b.v, not (C precedes D)`
- `within 2min before first A, B prevents C for 2s`

3.3 Informal semantics

3.3.1 Events

Events in the language have several characteristics:

- They are identified by their type, *e.g.* in the expression `absence_of A`, A is the type of the event.
- An event can be bound to a variable x like in `A x`.

- If bounded by a variable, a condition can be added on the event using the `where` construct: `A x where c`.

3.3.2 Patterns

A property defined using the language must contain at least one pattern. There two types of patterns: unary and binary patterns.

The first contains `occurrence_of n A` and its dual `absence_of A`. If n is not specified, it defaults to 1, else there should occur at least n! events A in th current scope.

The rest are three binary patterns: `A followed_by B`, `A precedes B` and `A prevents B`. They are very straightforward to understand by their names.

Examples of pattern satisfaction for various traces are given in table 3.1.

	$\langle A \rangle$	$\langle B \rangle$	$\langle A, A, C, B \rangle$	$\langle B, A \rangle$	$\langle A, B, A \rangle$
`absence_of A`	✗	✓	✗	✗	✗
`occurrence_of A`	✓	✗	✓	✓	✓
`occurrence_of 2 A`	✗	✗	✓	✗	✓
`A followed_by B`	✗	✓	✓	✗	✗
`A precedes B`	✓	✗	✓	✗	✓
`A prevents B`	✓	✓	✗	✓	✗

Table 3.1: Examples of pattern satisfaction for various traces [Ble17]

3.3.3 Scopes

Scopes are a mean to designate the range of a trace where a property should hold. They are delimited by optionally bound events. The scopes we propose can be classified according to their arity, *i.e.* the number of events types they expect. Unary scopes, illustrated in figure 3.2, are the basic building blocks.

An important consequence of the grammar definition is that scopes can be nested. For instance, `after last A, after each B, P` will hold if and only if P holds after each B occurring after the last occurrence of A. Nesting scopes properly allows defining more abstract scopes such as the two binary scopes illustrated in figure 3.3.

3.3.4 Timed Variants

Unary scopes and binary patterns may be additionally constrained with a duration expressed in common time units.

Both unary scopes (before and after) can be prefixed with the `within` keyword and a duration expression. Binary patterns, built upon unary scopes, may also be extended with a suffix and a duration expression.

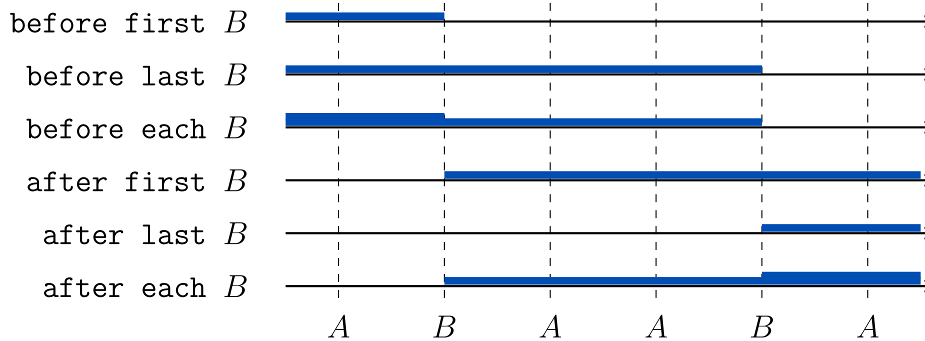

Figure 3.2: Graphical representation of the unary scopes [Ble17]

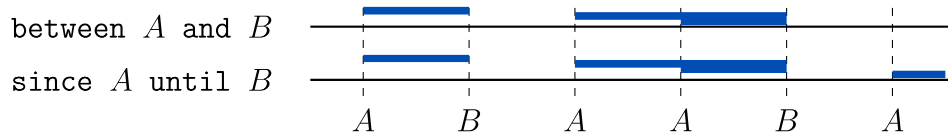

Figure 3.3: Graphical representation of the binary scopes [Ble17]

3.3.5 Quantifiers

The language uses quantified properties to exploit the values in event parameters. The universal quantifier takes the following form: `forall a in L, P`, where `a` is an identifier and `L` is a list. The existential quantifier (`exists`) is also defined as usual.

3.3.6 Event Selection

An expression takes the same syntactic form as scopes. It wraps another property that will be evaluated in a environment extended with the selected event. In the `each` case, the property must be true for all events matching the event descriptor.

3.4 Semantics

The semantics of PARTRAP are fully explained in the Appendix and thus will not be discussed.

— 4 —

A Disjunctive Based Coverage System for ParTraP

One coverage option to the properties was using existing tools. However, this requires translating the Haskell code into Java first, which would be hard to relate the Java code to ParTraP.

In Chapter 2, we introduced some well-known coverage criteria. However, due to the fact that we are dealing with temporal logic properties, most of them were not adapted to our problem. MCDC and Unique First Cause seemed to be good possible candidates. The first is very good in determinging the effect of each condition in the validity of the entire expression. The second was made specifically for temporal logic which is what our language is based on. Sadly both criteria came short for the below reasons.

The Unique Cause version of MCDC is very restricting. It prevents the coverage of very simple expressions which are very common and valid. Consider the property:

$$\texttt{not}\ (\texttt{occurrence_of}\ E1\ )\ \texttt{or}\ (\texttt{occurrence_of}\ E1\ \ \texttt{and}\ \texttt{occurrence_of}\ E2\ )$$

An event of type $E1$ is coupled with its negation, thus violating the rule of independent effect of conditions.

Unique First Cause, based on Masked MCDC, cares mainly about finding the first subexpression where a condition is validated. However, an expression may be validated in several ways. Reporting the reason back to the user would be problematic as it would be difficult to determine the exact cause of validating the expression using the same DSL. Consider the property:

$$\texttt{after first}\ E\,x\ \texttt{where}\ c,\ P \tag{4.1}$$

The expression can be satisfied in two ways:

- Absence of the event E
- Occurrence of the event E and after its first occurrence, the property P holds

Both of these disjuncts can be satisfied in a given trace, yet the user will only be notified of the first cause, first disjunct here. Thus, this method would obfuscate all other satisfiable disjuncts.

4.1 Using Disjunctive Normal Form - DNF

Our work is focused on linear temporal logic, thus according to Jianwen Li *et al.* [LPZ+13], an LTL formula may be represented using DNF if it is in the form of $\phi := \bigvee_i(\alpha_i \wedge X_{\phi_i})$, where α_i is a finite conjunction of literals, and $\phi_i := \bigwedge \phi_{ij}$ where ϕ_{ij} is either a literal, or an Until, Next or Release formula.

Inspired by the above, we have decided to break down the expressions written in ParTraP into a disjunction of conjunctions. Each disjunct on its own will be satisfying the original expression. Then, we will measure the number of statisfiable disjuncts over a set of trace files generated by the system. This will be the coverage criteria that we will be developing for this work.

This technique has several advantages:

- Keep the output within the constructs of ParTraP, this helps the user to easily relate to the results
- Identify the exact reason(s) of satisfaction of the original expression
- Identify test cases that were not covered by the testing phase of the system
- Identify weakly constructed properties while using ParTraP

4.2 Language Modifications

In example 4.1, we expressed two possible ways to satisfy the expression. Thus, we can rewrite the example within the same constructs of ParTraP to get the below:

$$\begin{aligned}&\texttt{after first } E\,x \texttt{ where } c\texttt{, } P = \\ &\texttt{not } (\texttt{occurrence_of } E\,x \texttt{ where } c) \\ &\ \texttt{or} \\ &((\texttt{occurrence_of } E\,x \texttt{ where } c) \texttt{ and } (\texttt{after first } E\,x \texttt{ where } c\texttt{, } P))\end{aligned} \tag{4.2}$$

The expression is divided into a disjunction of two expressions. Looking at the second disjunct, we notice a critical issue where the "`after first`" expression is repeated, this creates an infinite loop where ther is decomposing to itself in each iteration. This has lead us to introduce three new constructs to strengthen the language.

The first construct is `after!`, strict after. This is similar to the `after` construct in the fact that it requires the satisfiability of a property after the occurrence of the event. However, the difference is that the event must occur, whereas the non strict version can be satisfied with the absence of the event. It is defined below:

$$\frac{|M(\tau,(E_i,x_i)_{i=1}^n,c,\eta,o)| > 0 \quad \forall(j,k,\eta') \in M(\tau,(E_i,x_i)_{i=1}^n,c,\eta,o).\,(\tau_i)_{i>k} \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{ after! } o \texttt{ set(}E_1\,x_1\texttt{, } \ldots \texttt{, }E_n\,x_n\texttt{) where } c\texttt{, } P} \textsc{Aft!}$$

The second construct is `before!`, strict before. This is similar to the `after!` in the fact it adds an additional condition for the satisfiability with respect to the non strict version with the requirement of occurrence of the event. It is defined below:

$$\frac{|M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o)| > 0 \quad \forall (j,k,\eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, (\tau_i)_{i<k} \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{ before! } o \texttt{ set(}E_1\, x_1 \texttt{, ... ,} E_n\, x_n\texttt{) where } c\texttt{, } P} \textsc{Bfr!}$$

The final construct, `exists`, extends the grammar with the following dual of the `forall` construct:

$$\texttt{exists } x \texttt{ in } e\texttt{, } P_1 \texttt{ = not (forall } x \texttt{ in } e\texttt{, not } P_1\texttt{)}$$

After defining these three new constructs, we can how see the changes applied directly on example 4.2 to become 4.3 below:

$$\begin{aligned}
&\texttt{after first } E\, x \texttt{ where } c\texttt{, } P = \\
&\texttt{not (occurrence_of } E\, x \texttt{ where } c) \\
&\ \texttt{or} \\
&(\texttt{after! first } E\, x \texttt{ where } c\texttt{, } P)
\end{aligned} \tag{4.3}$$

In the modified version, we have merged `occurrence_of E` with `after first E` into an `after! first E`. This helps in avoiding the infinite loop problem of the `after` construct, and simplifies the disjunctive expression by removing useless terms.

4.3 Term Rewriting System

In order to rewrite the rules defined for ParTraP into disjunctive form, we used a term rewriting system. This system will help decompose our properties, similarly to example 4.3. It must preserve the semantics of the language. The decomposition is based on the rules defined in section 2.4. They are classical rules and axioms applied by pushing the negations inwards, and distributing the disjunctions over the conjunctions.

We write rules as `A` $\mapsto$ `B`. It means the expression `A` is rewritten into its new form `B`. `B` must have the same semantics as A, in other words, if τ is a trace, $\tau \vDash A \Leftrightarrow \tau \vDash B$ where $\vDash$ is the satisfaction relation. The full set of rewrite rules is given below. These rules do not follow any specific order and thus making the system nondeterministic.

Following classical logic, double negation can be eliminated:

$$\texttt{not not } P \mapsto P$$

Applying de Morgan's laws, negations can be pushed inwards as in the following couple of rules.

$$\texttt{not } (P_1 \texttt{ or } P_2) \mapsto (\texttt{not } P_1) \texttt{ and } (\texttt{not } P_2)$$

$$\texttt{not}\ (P_1\ \texttt{and}\ P_2) \mapsto (\texttt{not}\ P_1)\ \texttt{or}\ (\texttt{not}\ P_2) \tag{4.4}$$

The quantifiers `forall` and `exists` are complete duals, thus the negation of one is the other.

$$\texttt{not}\ (\texttt{forall}\ x\ \texttt{in}\ e,\ P) \rightarrow\ \texttt{exists}\ x\ \texttt{in}\ e,\ \texttt{not}\ P$$

$$\texttt{not}\ (\texttt{exists}\ x\ \texttt{in}\ e, P) \rightarrow \texttt{forall}\ x\ \texttt{in}\ e,\ \texttt{not}\ P$$

Because `absence_of` is simply defined as the negation of `occurrence_of`, we can simplify them:

$$\texttt{not}\ (\texttt{occurrence_of}\ E\ x\ \texttt{where}\ c) \mapsto \texttt{absence_of}\ E\ x\ \texttt{where}\ c$$

$$\texttt{not}\ (\texttt{absence_of}\ E\ x\ \texttt{where}\ c) \mapsto \texttt{occurrence_of}\ E\ x\ \texttt{where}\ c$$

The last two constructs are `after` and `before` along with their strict versions. `after!` is similar to it and will not be detailed, except for its negation. The same rewrite method can be applied to the `before` and its strict version as well, thus it will not be mentioned as well. `after` can be decomposed into two disjuncts:

$$\begin{aligned}\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P \mapsto\ & \texttt{not}\ (\texttt{occurrence_of}\ E\ x\ \texttt{where}\ c)\ \texttt{or} \\ & (\texttt{after!}\ \ \texttt{o}\ E\ x\ \texttt{where}\ c,\ P)\end{aligned} \tag{4.5}$$

This rule is similar to the one in example 4.3.

As seen in figure 3.3, a property is said to be satisfied when using the `after` construct if it happens at anytime after the occurrence of the event at hand. Since we have a disjunction between two properties, P_1 and P_2, then if any of them is satisfied, satisfies the initial unrefined property. Thus, the disjunction can be distributed here into two subexpressions:

$$\begin{aligned}\texttt{after o}\ E\ x\ \texttt{where}\ c,\ (P_1\ \texttt{or}\ P_2) \mapsto\ & (\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P_1)\ \texttt{or} \\ & (\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P_2)\end{aligned}$$

Similarly, `after` can be distributed over a conjunction:

$$\begin{aligned}\texttt{after o}\ E\ x\ \texttt{where}\ c,\ (P_1\ \texttt{and}\ P_2) \mapsto\ & (\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P_1)\ \texttt{and} \\ & (\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P_2)\end{aligned}$$

The next rule is the negation of the `after` construct:

$$\begin{aligned}\texttt{not}\ (\texttt{after o}\ E\ x\ \texttt{where}\ c,\ P) \mapsto\ & (\texttt{occurrence_of}\ E\ x\ \texttt{where}\ c)\ \texttt{and} \\ & \texttt{not}\ (\texttt{after!}\ \ \texttt{o}\ E\ x\ \texttt{where}\ c,\ P)\end{aligned} \tag{4.6}$$

To prove this rule, we apply the above rewrite rules. First we apply rule 4.5 on the expression between the parenthesis we get:

$$\texttt{not}\ (\texttt{not}\ (\texttt{occurrence_of}\ E\ x\ \texttt{where}\ c)\ \texttt{or}\ (\texttt{after!}\ \ \texttt{o}\ E\ x\ \texttt{where}\ c,\ P))$$

Then, pushing the negation inwards we end up with the two same disjuncts as in the rule.

The last rule presented is the negation of `after!` construct:

$$\texttt{not}\ (\texttt{after!}\ \ \texttt{o}\ E\ x\ \texttt{where}\ c,\ P) \mapsto \texttt{after o}\ E\ x\ \texttt{where}\ c,\ \texttt{not}\ P$$

This rules is a little complicated to understand from the first look. Consider the property without the negation. It is satisfiable if an event occurs and the property holds on the subtrace covered. To negate this, either the event never happens or the property was not satisfiable in the subtrace covered. This is exactly the definition of the `after` construct.

4.4 Example

Below we have a step-by-step example application of the term rewriting system applied on this property:

```
not (after first A, occurrence_of B and absence_of C)
```

We start by distributing the inner `after` over the conjunction by applying rule 4.4 to get:

```
not (after first A, occurrence_of B and after first A, absence_of C)
```

Then, we push the negation into the expression:

```
not (after first A, occurrence_of B) or not (after first A, absence_of C)
```

Now, we apply rule 4.6 on both subexpressions:

```
(occurrence_of A ) and not (after!  first A, occurrence_of B)

or

(occurrence_of A ) and not (after!  first A, absence_of C)
```

Finally, we push the negation inwards in the second part of both disjuncts:

```
(occurrence_of A ) and (after!  first A, absence_of B)
```

```
or

(occurrence_of A ) and (after!  first A, occurrence_of C)
```

As we can see, applying the rules defined for our rewriting system, we managed to rewrite the expression by applying them four times. The end result has the semantics as the initial expression. It also resulted in two disjunctive subexpressions that any of which may satisfy the original suggested property. We note here the existence of some redundancy with `occurrence_of A` that could be discarded. We further discuss this issue in the following chapter.

— 5 —

Implementation and Case Study

5.1 Implementation Procedure and Discussion

After defining the set of term rewrite rules, we decided to implement the rewrite system and test our theory on a test suite. To implement this system, we choose Haskell as our coding language since the DSL is already built using it, and it would be easier to integrate it with the language. Then we had to select a set of properties to test their satisfiability against the test suite. This suite contains a set of 100 trace files generated by the medical system used in previous surgeries that have taken place in the real world.

We chose six propeties that are commonly used to be satisfied. We started by generating their disjunctive subproperties using our term rewritng system. Then, we applied the resulting subproperties on a set of 100 trace files provided to us by the medical system company.

Property 1: The trace contains an event with a protocol 0.

```
occurrence_of EnterState v where v.state ==
"mainCasp.ReaquisitionProtocol0" ↦

occurrence_of EnterState v where (v.state) ==
(mainCasp.ReaquisitionProtocol0)
```

This property is simply not reducible, thus our term rewriting system was not applied here. It was 26% covered over the trace files, but such low coverage has no effect as it is allowed not to have $ReaquisitionProtocol0$.

Property 2: The trace does not contain at anypoint a record of having the temperature of the camera outside the normal/expected boundries.

```
absence_of Temp t where not (20.0 <= t.v1 and t.v1 < 50.0) ↦

absence_of Temp t where not (((20.0) <= (t.v1)) and ((t.v1) < (50.0)))
```

Again, this property can be rewritten by our rewrite system for it is not reducible. It was 74% covered which is high enough. However, if the second property was manually

edited, the tester will end up with several disjuncts (*i.e.* t.v1 >= 50 and t.v1 < 20), thus pinpointing the reason of failure in the 26% failing ones.

Also, it would be interesting to express a new property where we check whether the incorrect temperature has an effect on the quality of the final surgical result.

Property 3: The distance between pairs of hip centers is less than 1.0. Where `dist` : $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ is an external function returning the euclidean distance between two points.

```
after each HipCenter h1, absence_of HipCenter h2 where 1.0 <= dist(h1.point,
h2.point) ↦

absence_of HipCenter h1

or

after!  each HipCenter h1, absence_of HipCenter h2
where (1.0) <= (dist(h1.point, h2.point))
```

This property was divided into two disjuncts, the first was covered by 7% and the second by 92%. This raised some questions, why the first property was covered, and why is the total not 100% if they were all successful surgeries. After analyzing the trace files, we discovered the reason for covering the first disjunct was for the presence of shoulder surgeries in the trace files. These traces were of another MCPS provided by our partners by mistake. More interestingly, the missing 1% was found to be an actual error in the surgical procedure that was not detected while the system was running.

Same as before, it would be interesting to break the second disjunct into two disjuncts manually to consider the case where there is an absence of *HipCenter h*2, and the case where *HipCenter h*2 appears but with a satisfying distance predicate.

Property 4: After each action changing the state of the system, the next button should not be clicked before a minimum period of 0.5 seconds, where `ts` is the timestamp of the event.

```
after each EnterState e, absence_of ActionNext a where a.ts - e.ts < 0.5
↦

absence_of EnterState e

or

after!  each EnterState e, absence_of ActionNext a
where ((a.ts) - (e.ts)) < (0.5)
```

This property was divided into two disjuncts. The first disjunct was not covered, and the second was only 53%. This value was a bit alarming for how little it was. After analyzig the traces and discussing the developers of the system, we discovered two possibilities:

- Users became familiar with the system and click on skip buttons quickly
- Users sometimes tend to double click the buttons on screen

The first reason could affect the user by missing critical information and thus we agreed that there should be a small load period between screens. The second reason was riskier, as one might accidentally increase/decrease certain parameters unexpectedly, thus one might prevent triggering the event if the previous event was less than the threshold.

Property 5: The camera should be connected prior to the connection of the trackers.

```
CameraConnected precedes EnterState e where e.state ==
"mainCasp.TrackingConnection.TrackersConnection" ↦

absence_of EnterState e where (e.state) ==
(mainCasp.TrackingConnection.TrackersConnection)

or

before!  each EnterState e where (e.state) ==
(mainCasp.TrackingConnection.TrackersConnection),
occurrence_of CameraConnected
```

Again, this property was divided into two disjuncts. The first one was 6% covered, and the second 94% covered. The first part was only covered due to the usage of shoulder surgeries, but it has no effect on the overall outcome. This means that this property was fully covered.

Property 6: During the search for trackers and until the trackers are connected, all tracker types should be detected.

```
between SearchTrackers st and ExitState e where e.state ==
"mainCasp.TrackingConnection.TrackersConnection",
forall ty in st.types, occurrence_of TrackerDetected td
where td.ty == ty ↦
absence_of SearchTrackers st

or

after!  each SearchTrackers st, absence_of ExitState e
where (e.state) == mainCasp.TrackingConnection.TrackersConnection

or

after!  each SearchTrackers st, before!  first ExitState e
where (e.state) == mainCasp.TrackingConnection.TrackersConnection,
forall ty st.types, occurrence_of TrackerDetected td
```

```
where (td.ty) == (ty)
```

This property was divided into three disjuncts. The first disjunct was 0% covered, the second was 6% covered, and the third was 84% covered. Again the coverage of the second disjunct was due to the shoulder surgery included in the trace files. What was interesting is the fact that the total coverage was 10% short from full coverage.

Analyzing the property showed that it was weakly structured, as it considers tracker types that exist between current tracker connected and the exit state. This method discards all previously registered tracker types. To solve this problem, we introduced a new construct named *given* updating the grammar of the language to become:

$$\begin{array}{lcl} \langle prop \rangle & ::= & \langle pattern \rangle \\ & | & \ldots \\ & | & \text{`}\texttt{given}\text{'} \ (\text{`}\texttt{each}\text{'} \mid \text{`}\texttt{first}\text{'} \mid \text{`}\texttt{last}\text{'}) \ \langle event \rangle \ \text{`}\texttt{,}\text{'} \ \langle prop \rangle \\ & | & \ldots. \end{array}$$

Figure 5.1: Updated syntax of the proposed DSL

The full modified grammar can be found in the Appendix. Then, we introduced the following rule:

$$\frac{\forall (j,k,\eta') \in M(\tau, (E_i, x_i)_{i=1}^{n}, c, \eta, o).\, \tau \vDash_{\eta'} P}{\tau \vDash_{\eta} \texttt{given}\ o\ \texttt{set(}E_1\ x_1\texttt{, ... ,}E_n\ x_n\texttt{)}\ \texttt{where}\ c\texttt{,}\ P} \textsc{Given}$$

The rule for the `given` expression is the same as AftT without the range restriction. The full modified semantic rules can be found in the Appendix.

We then modified the property at hand with this new construct which results in the below:

```
before each EnterState e
where e.state == "mainCasp.TrackingConnection.TrackersVisibCheck",
given last SearchTrackers st,
forall ty in st.types,
occurrence_of TrackerDetected td where td.ty == ty
```

5.1.1 Discussion

As a result of applying the disjunctive rewrite rules, we managed to reach several goals:

- Validate the original properties are satisfiable and detect possible defects in the implementation of the language
- Test if satisfying at least one of our disjunct terms satisfies the original property, thus proving the effectiveness of our rewriting system

- Understand which disjuncts are more satisfiable. This helps the engineers pointing out the accurate reason of satisfiability

5.2 Side Effects of the System

We noticed some defects in the generation of the disjunctive rules, where rewriting a property might generate disjunct expressions that contain some redundancy, or generating unsatisfiable disjuncts.

5.2.1 Example 1

Consider the following property:

```
(after first A, absence_of C) and (after each B, absence_of A)
```

We applied our rewriting system on the above property to get one of its resulting disjuncts below:

```
(absence_of A) and (after!  each B, absence_of A)
```

This disjunct uses `absence_of A` in both sides of the conjunctive expression. It would have been simpler to have the property below:

```
(absence_of A) and (occurrence_of B)
```

This will not affect the overall output of the coverage and thus it is useless for now to modify it. However, it would be better to do it in the future to make the output readability simpler.

5.2.2 Example 2

Consider the following property:

```
(after first A, absence_of C) and
(after each B, after first A, before first B, absence_of C)
```

We applied our rewriting system on the above property to get one of its resulting disjuncts below:

```
(absence_of A) and
(after!  each B, after!  first A, before!  first B, absence_of C)
```

This disjunct uses `absence_of A` and `after!  first A` in both sides of the conjunctive expression. The second subexpression specifically requires an occurrence of the event A, however, the first subexpression clearly states the absence of this event. This means if

the first subexpression is $true$ then the second will definitely be $false$ and the property will not be satisfiable. Same if the first was $false$ then no matter what was the second, it will not be satisfiable. Thus, such property is impossible to satisfy.

Further work should be made in order to prevent the evaluation of the unsatisfiable expressions. This is more detailed in the next chapter.

— 6 —
Working with SMT, Preliminary Work

6.1 Introduction SMT Solvers

We have seen in the previous chapter that we might end up with impossible cases when generating the disjunctive terms. These cases should be discarded or marked as impossible so that the verification engineers don't try to find a test case that verifies them. One solution is to use SAT (Boolean satisfiability problem) solvers. SAT is given a Boolean formula to check whether it is satisfiable by assigning the proper logical values to make it $true$.

Due to the fact that SAT solvers work on the Boolean logic level, and most systems are designed to function with a higher level, it is expensive to use them. Thus, an extension of SAT was introduced named Satisfiability Modulo Theories (SMT). SMT uses a higher level language which is first-order logic [EE01]. It is efficient in handling of many useful theories arising in software verification. One of its advantages is its ability to handle linear arithmetic. Scientists tend to use such solvers in their verification applications because they are efficient, automatic, and powerful.

The example in figure 6.1 demonstrates a valid Boolean formula since for all values of x, a value of y can be found satisfying it. For example, if x was $false$, then if y is set to $true$ the expression is satisfied, this is true if the values are switched.

$$\forall \texttt{x}\ \exists \texttt{y}\ \ (x \vee y) \wedge (\neg x \vee \neg y)$$

Figure 6.1: Example of satisfiable Boolean formula

We decided to use SMT solvers for their usage of first order logic, because we can not translate our expression into Boolean logic. In addition, these solvers use universal and existential quantifiers and permit the usage of constant and function symbols.

There are several engines available as SMT solvers. One solver is *Z3* by Microsoft Research under the MIT license [Z3]. We decided to use this engine, through the SBV Haskell library, due to its support for multiple platforms (Linux, Mac OS, Windows, FreeBSD). In addition, it allows us to write *Z3* formulas directly in Haskell, which would

make easier for us to integrate into our work. Despite that it sounds convenient, we fell in many undocumented traps.

6.2 Translation From ParTraP to First Order Logic

In order to use such SMT solver, we had to introduce a set of rewrite rules that would be applied to each disjunct resulting from our implementation. Then, only the ones that are proven satisfiable are passed into the testing phase to find their coverage.

The predicate $\mathit{matches} : \mathit{Event} \times \Sigma \times \mathit{Env} \times \mathit{Var} \times \mathit{Expr} \rightarrow \mathbb{B}$ holds when an event matches a description and is defined as follows:

$$\mathit{matches}(e, E, \eta, x, c) = (\mathit{name}(e) = E) \land (\eta[x \mapsto \mathit{parameters}(e)] \vdash c \downarrow \textbf{true})$$

This definition of *matches* comes prior to the updates introduced to the language, thus the translation lacks the support of sets that the newly introduced function M possesses, presented earlier in Chapter 3.

The predicate $\mathit{proposition} : \mathit{Trace} \times \mathit{Env} \times \mathit{Property} \rightarrow \mathbb{B}$ holds when the property holds over the given trace and environment.

We present the set of transformation rules from ParTraP to first-order logic below.

The application of the predicate over a negation of a property is equivalent to negating its application on the property.

$$\mathit{proposition}(\tau, \eta, \texttt{not}\ P) = \neg\mathit{proposition}(\tau, \eta, P)$$

The application of the predicate over a disjunction/conjunction of properties is equivalent to the application of it to each disjunct/conjunct term.

$$\mathit{proposition}(\tau, \eta, P_1\ \texttt{or}\ P_2) = \mathit{proposition}(\tau, \eta, P_1) \lor \mathit{proposition}(\tau, \eta, P_2)$$

$$\mathit{proposition}(\tau, \eta, P_1\ \texttt{and}\ P_2) = \mathit{proposition}(\tau, \eta, P_1) \land \mathit{proposition}(\tau, \eta, P_2)$$

The satisfiability of the predicate applied to the `occurrence_of` construct is equivalent to finding a subtrace with a matching event.

$$\mathit{proposition}(\tau, \eta, (\texttt{occurrence_of}\ E\ x\ \texttt{where}\ c)) = \exists i\ \mid \mathit{matches}(\tau_i, E\ , \eta, x, c)$$

The same holds for the `absence_of` construct.

$$\mathit{proposition}(\tau, \eta, (\texttt{absence_of}\ E\ x\ \texttt{where}\ c)) = \nexists i\ \mid \mathit{matches}(\tau_i, E\ , \eta, x, c)$$

The last two constructs are `after` and `before` along with their strict versions. The same rewrite method can be applied to the `before` and its strict version, thus it will not be mentioned as well.

The predicate is satisfied over the property `after first` when there is either no match of such event, or there is a match (not preceded by another match) and the predicate holds on the subtrace starting from the match location.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after first } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\nexists i \mid \mathit{matches}(\tau_i, E,\eta,x,c)) \;\vee\; (\exists i \mid \mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge \\ &(\nexists j \mid (j < i \;\wedge\; (\mathit{matches}(\tau_j,E,\eta,x,c))) \;\wedge \\ &\mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

The predicate is satisfied over the property `after! first` for the same conditions as the non strict version, except that there should always be a match of the event.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after! first } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\exists i \mid \mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge\; (\nexists j \mid (j < i \;\wedge\; (\mathit{matches}(\tau_j,E,\eta,x,c))) \;\wedge \\ &\mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

The predicate is satisfied over the property `after last` when there is either no match of such event, or there is a match (not followed by another match) and the predicate holds on the subtrace starting from the match location.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after last } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\nexists i \mid \mathit{matches}(\tau_i, E,\eta,x,c)) \;\vee\; (\exists i \mid \mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge \\ &(\nexists j \mid (j > i \;\wedge\; (\mathit{matches}(\tau_j,E,\eta,x,c))) \;\wedge \\ &\mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

The predicate is satisfied over the property `after! last` for the same conditions as the non strict version, except that there should always be a match of the event.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after! last } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\exists i \mid \mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge\; (\nexists j \mid (j > i \;\wedge\; (\mathit{matches}(\tau_j,E,\eta,x,c))) \;\wedge \\ &\mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

The predicate is satisfied over the property `after each` when there is either no match of such event, or whenever there is a match the predicate holds on the subtrace starting from the match location.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after each } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\nexists i \mid \mathit{matches}(\tau_i, E,\eta,x,c)) \;\vee \\ &(\forall i \mid (\mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge\; \mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

The predicate is satisfied over the property `after! each` for the same conditions as the non strict version, except that there should always be a match of the event.

$$\begin{aligned}&\mathit{proposition}(\tau,\eta,(\texttt{after! each } E\; x \texttt{ where } c\texttt{, } P)) = \\ &(\forall i \mid (\mathit{matches}(\tau_i, E,\eta,x,c) \;\wedge\; \mathit{proposition}((\tau_j)_{j>i},\eta[x\mapsto \mathit{parameters}(\tau_i)],P)))\end{aligned}$$

These implemented these rules in the Haskell API of the SMT solver. The process of coverage now starts by creating the set of disjunctive subproperties, then passing each of them through these SMT rewrite rules, finally only the satisfiable ones are used for measuring the coverage over the test suite.

6.3 Example and Discussion

Below we will demonstrate an example application of the transformation rules from ParTraP to first order logic applied on this property:

```
after first A , before first B , absence_of C
 and
after first B , absence_of C
```

We start by applying the term rewriting rules introduced in Chapter 4 to get:

```
absence_of A                                                         (6.1)
or
(after!  first A, absence_of B) and (after!  first A, absence_of B)  (6.2)
or
(after!  first A, absence_of B) and
(after!  first A, before!  first B, absence_of B)                    (6.3)
or
(after!  first A, absence_of B) and
(after!  first A, before!  first B, after!  first B, absence_of C)   (6.4)
or
(after!  first A, before!  first B, absence_of C) and
(after!  first A, absence_of B)                                      (6.5)
or
(after!  first A, before!  first B, absence_of C) and
(after!  first A, before!  first B, absence_of B)                    (6.6)
or
(after!  first A, before!  first B, absence_of C) and
(after!  first A, before!  first B, after!  first B, absence_of C)   (6.7)
```

Then we apply the translation rules, introduced in this chapter, to transform the above disjuncts into first order logic and check their satisfiability through the SMT solver. The results indicate that properties 6.3, 6.4, 6.5, and 6.6 are all unsatisfiable. We can see why in details below.

Property 6.3, second conjunct, states that must exist a `B` based on the rule of `before!` construct. However, it continues to state the absence of it proving its unsatisfiability. The same can be said about properties 6.5 and 6.6.

Property 6.4 requires the absence of event `B` in the first conjuct, then negates it by using the `before!` construct.

However, a surprising result was that it discarded both properties 6.2 and 6.7 as they are not satisfiable using the SMT solver due to the redundancy in them, even though they are both satisfiable. In 6.2, the conjunct is duplicated while in 6.7, the second conjunct is reducible to:

```
(after!  first A, absence_of C)
```
(6.8)

which makes the first conjunct a redundant one.

— 7 —
Future Work and Conclusion

7.1 Conclusion

A new language called ParTraP has been developed to adapt formal methods to verify safety properties of medical systems. However, it remained to define a coverage metric for the trace properties. Such metric aims at pinpointing the exact cause of satisfiability of such properties. In addition, it measures the overall coverage of a property in test suite. This metric should preserve the semantics of the language to simplify its usage by the users by not introducing a completely different tool. This will minimize the cost of training and testing for such systems.

In this report, we proposed to break down the properties expressed over ParTraP into a set of disjunctive subproperties. Each subproperty is sufficient on its own to satisfy the original property. For this purpose, we introduced a term rewriting system in Chapter 4 along with a step by step example of its application. This system required the modification of ParTraP to adapt new constructs that would improve its expressiveness. These constructs were: `after!`, `before!`, and `given`. The system is then tested over six properties identified as default (should always be satisfiable). The resulting disjuncts are then applied over a test suite composed of trace files gathered by real world application of the MCPS. These positive results demonstrate the effectiveness of such a system.

Further work has then been introduced by creating a set of transformation rules from ParTraP into first-order logic. This transformation was necessary to test the disjuncts using SMT solvers to verify their potential satisfiability. This work is in an advanced stage, to be completed in the near future.

7.2 Future Work

Our preliminary results of applying the transformation rules to first-order logic presented in Chapter 6 are very promising, yet further improvements are needed to increase their effectiveness. One important improvement is to extend the rules to try to detect redundancies and optimize them in a way preventing the SMT solver from refuting them.

Currently, guards are not covered in both of our term rewriting system and SMT rules implementation. However, they can control the satisfiability of a property, for example, a disjunct of two terms with a guard can break down a property into two. This would insure further decomposition of the properties and an in depth analysis of their satisfiability. An example decomposition is the one discussed in 5, Property 2.

Further interesting work would be generating traces from the resulting disjuncts satisfying them. This can be done using the SMT solver as it can be configured to output example valuations satisfying the property. These traces can be represented within a graphical user interface. This would simplify the system for the users even more, especially when the property is still complex after it has been decomposed.

Finally, we will include both of the systems generated in this work in the toolset of ParTraP making it easier for users to develop and test their properties using the same tool.

Bibliography

[BBdB+16] Fabrice Bertrand, Yoann Blein, Lydie du Bousquet, Roland Groz, Yves Ledru, and Arnaud Clère. Modmed wp6/d1: Requirements analysis. In *Technical report,BlueOrtho*. LIG, MinMaxMedical, 2016.

[Bei90] Boris Beizer. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 2nd edition, 1990.

[BKRS12] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. Ltl to büchi automata translation: Fast and more deterministic. *CoRR*, abs/1201.0682, 2012.

[Ble17] Yoann Blein. Wp1/d1: Preliminary definition of a domain specific specification language. LIG, May 29, 2017.

[Chi94] S.P. Chilenski, J.J. – Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal, 9(5)*, 54(9):193–200, 1994.

[Cor11] Cornett, s. 2011. code coverage analysis. `http://www.bullseye.com/coverage.html`, 2011. Accessed: 2017-05-30.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[DP90] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, 1990.

[EE01] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Academic press, 2001.

[FW09] Gordon Fraser and Franz Wotawa. Complementary criteria for testing temporal logic properties. In *Proceedings of the 3rd International Conference on Tests and Proofs*, TAP '09, pages 58–73, Berlin, Heidelberg, 2009. Springer-Verlag.

[Gen15] MODel-Based Verification of MEDical Cyber-Physical Systems, 2015. Generic call for proposals.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to BüChi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 53–65, London, UK, UK, 2001. Springer-Verlag.

[KJDSJJLK01] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[KP13] MIKHAIL KALKOV and DZMITRY PAMAKHA. Code coverage criteria and their effect on test suite qualities. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 2013.

[LPZ+13] Jianwen Li, Geguang Pu, Lijun Zhang, Zheng Wang, Jifeng He, and Kim Guldstrand Larsen. *On the Relationship between LTL Normal Forms and Büchi Automata*, pages 256–270. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Mod15] Modmed goals and methodology. `https://sites.google.com/a/minmaxmedical.com/modmed/goals-and-methodology`, 2015. Accessed: 2017-06-07.

[MSW14] Dominique Méry, Bernhard Schätz, and Alan Wassyng. The Pacemaker Challenge: Developing Certifiable Medical Devices (Dagstuhl Seminar 14062). *Dagstuhl Reports*, 4(2):17–37, 2014.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[Nta88a] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, June 1988.

[Nta88b] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Software Eng.*, 14(6):868–874, 1988.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[RB11] EUROCAE/ED-12B RTCA/DO-178B. Software considerations in airborne systems and equipment certification, rtca, December 2011.

[Rop94] Marc Roper. *Software Testing*. McGraw-Hill Book Company, 1994.

[Tea01] Certification Authorities Software Team. Rationale for accepting masking mc/dc in certification projects. Technical report, August 2001.

[TJD+15] Safouan Taha, Jacques Julliand, Frédéric Dadeau, Kalou Cabrera Castillos, and Bilal Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Aspects of Computing*, 27(4):641–664, dec 2015.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 25–36, New York, NY, USA, 2006. ACM.

[Z3] Z3 prover, open source project. `https://github.com/z3prover/z3/wiki`. Accessed: 2017-05-30.

— A — Appendix

A.1 ParTraP Semantics

A.1.1 Preliminary Definitions

If (e_i) is an untimed trace, (σ_k, x_k) is a sequence of event names and variable names, c is a condition expression and η is an environment, let

$$M_{\text{all}}((e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta) = \{(i_k)_{k=1}^n \mid \mathit{name}(e_{i_k}) = \sigma_k \wedge \eta' \vdash c \downarrow \textbf{true}\},$$

where

$$\eta' = \eta[x_1 \mapsto \mathit{param}(e_{i_1}), \ldots, x_n \mapsto \mathit{param}(e_{i_n})], \tag{A.1}$$

be the set of index sequences of (e_i) such that each event sequence *matches* the event names (σ_k), and respects the condition c when evaluated in the environment η extended with the event parameters. Put simply, M_{all} computes all the event sets matching an event set description for a given trace and a given environment.

Let S be a finite set of sequences of equal length. We write $\min S$ (resp. $\max S$) to denote the minimal (resp. maximal) element of S in colexicographic order, which is a variant of the lexicographical order obtained by comparing sequences from the right to the left. If o is an occurrence specifier, i.e. an element of $\{\texttt{first}, \texttt{last}, \texttt{each}\}$,

$$\mathit{select}(o, S) = \begin{cases} S & \text{if } o = \texttt{each} \\ \{\min\ S\} & \text{if } o = \texttt{first} \text{ and } S \neq \emptyset \\ \{\max\ S\} & \text{if } o = \texttt{last} \text{ and } S \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

restricts S according to the occurrence specifier o. The colexicographic ensures that the first element is the event set with the earliest last event and that the last element is the event set with the latest first event. This order is total for S, which guarantees the existence and uniqueness of the minimal and maximal elements.

If (t_i, e_i) is a trace, (σ_k, x_k) is a sequence of event names and variable names, c is a condition expression, η is an environment and o an occurrence specifier, let

$$M((t_i, e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta, o) = \\ \{(\min_k i_k, \max_k i_k, \eta') \mid (i_k)_{k=1}^n \in select(o, M_{\text{all}}((e_i)_{i=1}^m, (\sigma_k, x_k)_{k=1}^n, c, \eta))\},$$

where η' is defined as in (A.1), be the set of triplets that, for each event set matching the given description and occurrence specifier, is composed of the beginning index of the match in the trace, its ending index and the updated environment with the matched event parameters.

Finally, if (e_i, t_i) is a trace and l a natural, the function

$$upto((e_i, t_i)_{i=1}^n, l) = (e_i, t_i)_{i=1}^{max(\{j|t_j<l\}\cup n)}$$

slices the trace (e_i, t_i) from its beginning and up to the time limit l.

A.1.2 Semantics Rules

Properties are evaluated over finite traces and in a specific environment. The satisfaction relation between a trace τ, an environment η and a property p is the smallest relation $\tau \vDash_\eta p$ satisfying the following rules:

$$\frac{\tau \nvDash_\eta P}{\tau \vDash_\eta \texttt{not } P}\ \textsc{Neg} \qquad \frac{\tau \vDash_\eta P_1 \lor \tau \vDash_\eta P_2}{\tau \vDash_\eta P_1 \texttt{ or } P_2}\ \textsc{Disj}$$

$$\frac{\eta \vdash e \downarrow L \quad \forall v \in L.\, \tau \vDash_{\eta[x \mapsto v]} P}{\tau \vDash_\eta \texttt{forall } x \texttt{ in } e\texttt{, } P}\ \textsc{Forall}$$

$$\frac{\eta \vdash n_e \downarrow n \quad n \leq |M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, \texttt{each})|}{\tau \vDash_\eta \texttt{occurrence_of } n_e \texttt{ set(}E_1\ x_1\texttt{, } \ldots \texttt{,} E_n\ x_n\texttt{) where } c}\ \textsc{Occ}$$

$$\frac{\forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, (\tau_i)_{i>k} \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{ after } o \texttt{ set(}E_1\ x_1\texttt{, } \ldots \texttt{,} E_n\ x_n\texttt{) where } c\texttt{, } P}\ \textsc{Aft}$$

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, upto((\tau_i)_{i>k}, t_0 + \delta) \vDash_{\eta'} P}{\tau \vDash_\eta \texttt{within } \delta_e \texttt{ after } o \texttt{ set(}E_1\ x_1\texttt{, } \ldots \texttt{,} E_n\ x_n\texttt{) where } c\texttt{, } P}\ \textsc{AftT}$$

The rules Bef and BefT for the `before` scope are symmetrical to Aft and AftT, respectively, and are omitted here. We say that a trace τ *satisfies* a property P when $\tau \vDash_{[]} P$.

The first two rules are straightforward. The next one, Forall, handles universally quantified properties by first evaluating the expression that represents the quantification domain, and then evaluating the subsequent property for all values in that domain. Occ,

the only non-recursive rule, asserts the occurrence of a particular event set description by counting the number of event sets that match this description in the current trace.

The rules for the `after` scope are AFT, and its timed variant AFTT. They rely on the results of the M function to slice the trace after the end of each event set matching the description and to update the evaluation environment. Additionally, AFTT evaluates a duration expression and slices the trace so that it lasts at most for this duration.

The previous rules allow defining the additional logical expressions with the usual identities:

- P_1 `and` $P_2 \equiv$ `not` (`not` P_1 `or` `not` P_2)
- P_1 `implies` $P_2 \equiv$ `not` P_1 `or` P_2
- P_1 `equiv` $P_2 \equiv$ (P_1 `implies` P_2) `and` (P_2 `implies` P_1),

and the additional temporal expressions:

- `absence_of` $E \equiv$ `not` (`occurrence_of` E)
- A `followed_by` B `within` $\delta \equiv$
  `within` δ `after each` A, `occurrence_of` B
- A `precedes` B `within` $\delta \equiv$
  `within` δ `before each` B, `occurrence_of` A
- A `prevents` B `within` $\delta \equiv$
  `within` δ `after each` A, `absence_of` B
- `between` A `and` B, $P \equiv$ `after each` A, `before first` B, `P`
- `since` A `until` B, $P \equiv$
  (`between` A `and` B, P) `and` (`after last` B, `after first` A, P)

Finally, simple event descriptions are translated into event set descriptions with a single event, unbound events are bound to the empty variable name, and omitted guards defaults to `true`.

A.2 Modified ParTraP Grammar

⟨*prop*⟩ ::= ⟨*pattern*⟩
| ⟨*scope*⟩ ‘,’ ⟨*prop*⟩
| (‘forall’ | ‘exists‘) <ident> ‘in’ ⟨*expr*⟩ ‘,’ ⟨*prop*⟩
| ‘given’ (‘each’ | ‘first’ | ‘last’) ⟨*event*⟩ ‘,’ ⟨*prop*⟩
| ‘(’ ⟨*prop*⟩ ‘)’
| ‘not’ ⟨*prop*⟩
| ⟨*prop*⟩ (‘and’ | ‘or’ | ‘equiv’ | ‘implies’) ⟨*prop*⟩

⟨*scope*⟩ ::= [‘within’ ⟨*duration*⟩] (‘after’ | ‘before’) (‘each’ | ‘first’ | ‘last’) ⟨*event*⟩
| ‘between’ ⟨*event*⟩ ‘and’ ⟨*event*⟩
| ‘since’ ⟨*event*⟩ ‘until’ ⟨*event*⟩

⟨*pattern*⟩ ::= ‘absence_of’ ⟨*event*⟩
| ‘occurrence_of’ ⟨*expr*⟩ ⟨*event*⟩
| ⟨*event*⟩ ‘followed_by’ ⟨*event*⟩ [‘within’ ⟨*duration*⟩]
| ⟨*event*⟩ ‘precedes’ ⟨*event*⟩ [‘within’ ⟨*duration*⟩]
| ⟨*event*⟩ ‘prevents’ ⟨*event*⟩ [‘for’ ⟨*duration*⟩]

⟨*event*⟩ ::= ⟨*ident*⟩ [⟨*ident*⟩ [‘where’ ⟨*expr*⟩]]
| ‘set’ ‘(’ ⟨*ident*⟩ [⟨*ident*⟩] (‘,’ ⟨*ident*⟩ [⟨*ident*⟩])* ‘)’ [‘where’ ⟨*expr*⟩]

⟨*duration*⟩ ::= ⟨*expr*⟩ (‘ms’ | ‘s’ | ‘min’ | ‘h’ | ‘d’)

Figure A.1: Syntax of the modified DSL

A.3 Modified ParTraP Semantic Rules

$$\frac{\tau \not\models_\eta P}{\tau \models_\eta \texttt{not } P}\ \textsc{Neg} \qquad \frac{\tau \models_\eta P_1 \vee \tau \models_\eta P_2}{\tau \models_\eta P_1 \texttt{ or } P_2}\ \textsc{Disj}$$

$$\frac{\eta \vdash e \downarrow L \quad \forall v \in L.\, \tau \models_{\eta[x\mapsto v]} P}{\tau \models_\eta \texttt{forall } x \texttt{ in } e\texttt{, } P}\ \textsc{Forall}$$

$$\frac{\eta \vdash n_e \downarrow n \quad n \leq |M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, \texttt{each})|}{\tau \models_\eta \texttt{occurrence_of } n_e \texttt{ set(} E_1\ x_1 \texttt{, } \ldots \texttt{,} E_n\ x_n \texttt{) where } c}\ \textsc{Occ}$$

$$\frac{\forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, (\tau_i)_{i>k} \models_{\eta'} P}{\tau \models_\eta \texttt{ after } o \texttt{ set(} E_1\ x_1 \texttt{, } \ldots \texttt{,} E_n\ x_n \texttt{) where } c\texttt{, } P}\ \textsc{Aft}$$

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, upto((\tau_i)_{i>k}, t_0 + \delta) \models_{\eta'} P}{\tau \models_\eta \texttt{within } \delta_e \texttt{ after } o \texttt{ set(} E_1\ x_1 \texttt{, } \ldots \texttt{,} E_n\ x_n \texttt{) where } c\texttt{, } P}\ \textsc{AftT}$$

$$\frac{\forall (j, k, \eta') \in M(\tau, (E_i, x_i)_{i=1}^n, c, \eta, o).\, \tau \models_{\eta'} P}{\tau \models_\eta \texttt{given } o \texttt{ set(} E_1\ x_1 \texttt{, } \ldots \texttt{,} E_n\ x_n \texttt{) where } c\texttt{, } P}\ \textsc{Given}$$