ParTraP: A Language for the Specification and Runtime Verification of Parametric Properties

Yoann Blein

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Yoann BLEIN

Thèse dirigée par **Yves LEDRU**, Professeur Université Grenoble Alpes
et codirigée par **Lydie DU BOUSQUET**, Professeur Université Grenoble Alpes
préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

ParTraP : Un langage pour la spécification et vérification à l'exécution de propriétés paramétriques

ParTraP: A Language for the Specification and Runtime Verification of Parametric Properties

Thèse soutenue publiquement le **15 avril 2019**,
devant le jury composé de :

Monsieur SADDEK BENSALEM
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Président
Monsieur JULIEN SIGNOLES
INGENIEUR CHERCHEUR, CEA LIST - GIF-SUR-YVETTE, Rapporteur
Madame VIRGINIE WIELS
MAITRE DE RECHERCHE, ONERA CENTRE MIDI-PYRENEES, Rapporteur
Monsieur JEAN-MARC JEZEQUEL
PROFESSEUR, UNIVERSITE RENNES 1, Membre
Madame LYDIE DU BOUSQUET
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Co-directrice de thèse
Monsieur YVES LEDRU
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse

PARTRAP: A LANGUAGE FOR THE SPECIFICATION AND RUNTIME VERIFICATION OF PARAMETRIC PROPERTIES

Design, Semantics and Case Study

YOANN BLEIN

February 2019

Yoann Blein: ParTraP*: A Language for the Specification and Runtime Verification of Parametric Properties,* Design, Semantics and Case Study,

SUPERVISORS:
Yves Ledru
Lydie du Bousquet

ABSTRACT

Runtime verification is a promising technique to improve the safety of complex systems. These systems can be instrumented to produce execution traces enabling us to observe their usage in the field. A significant challenge is to provide software engineers with a simple formal language adapted to the expression of their most important requirements. In this thesis, we focus on the verification of medical devices. We performed a thorough analysis of a worldwide-used medical device in order to identify those requirements, as well as the precise nature of its execution traces. In the light of this study, we propose ParTraP, a formally defined language dedicated to property specification for finite traces. It is designed to be accessible to software engineers with no training in formal methods thanks to its simplicity and declarative style. The language extends the specification patterns originally proposed by Dwyer et al. with parametrized constructs, nested scopes, real-time and first-order quantification. We also propose a coverage measurement technique for ParTraP, and we show that coverage information provides insights on a corpus of traces as well as a deeper understanding of temporal properties. Finally, we describe the implementation of an Integrated Development Environment for ParTraP, which is available under a free and open-source license.

RÉSUMÉ

La vérification à l'exécution est une technique prometteuse pour améliorer la sûreté des systèmes complexes. Ces systèmes peuvent être instrumentés afin qu'ils produisent des traces d'exécution permettant d'observer leur utilisation dans des conditions réelles. Un défi important est de fournir aux ingénieurs logiciel un langage formel simple adapté à l'expression des exigences les plus importantes. Dans cette thèse, nous nous intéressons à la vérification de dispositifs médicaux. Nous avons effectué l'analyse approfondie d'un dispositif médical utilisé mondialement afin d'identifier les exigences les plus importantes, ainsi que la nature précise des traces d'exécution qu'il produit. À partir de cette analyse, nous proposons ParTraP, un langage défini formellement et dédié à la spécification de propriétés sur des traces finies. Il a été conçu pour être accessible à des ingénieurs logiciels non qualifiés en méthodes formelles grâce à sa simplicité et son style déclaratif. Le langage étend les patrons de spécification initialement proposé par Dwyer et al. avec des opérateurs paramétriques et temps-réel, des portées emboîtables, et des quantificateurs de premier ordre. Nous proposons également une technique de mesure de couverture pour ParTraP, et montrons que le niveau de couverture d'une propriété temporelle permet de mieux la comprendre, ainsi que le jeu de traces sur lequel elle est évaluée. Finalement, nous décrivons l'implémentation d'un environnement de développement intégré pour ParTraP, qui est disponible sous une licence libre.

PREFACE

This thesis presents my research conducted in the VASCO team at Laboratoire d'Informatique de Grenoble, to pursue a PhD in Computer Science from the doctoral school "Mathématiques, Sciences et Technologies de l'Information, Informatique" of the Université Grenoble-Alpes. My research activities have been supervised by Yves Ledru and Lydie du Bousquet, both professors at the Université Grenoble-Alpes, and funded by the *Agence Nationale de la Recherche* (ANR-15-CE25- 0010).

This work led to the following publications:

[1] Yoann Blein, Arnaud Clere, Fabrice Bertrand, Yves Ledru, Roland Groz, and Lydie du Bousquet. "Improving Trace Generation and Analysis for Medical Devices." In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017, Prague, Czech Republic, July 25-29, 2017*. IEEE, 2017, pp. 599–600. DOI: 10.1109/QRS-C.2017.135.

[2] Yoann Blein, Yves Ledru, Lydie du Bousquet, and Roland Groz. "Extending specification patterns for verification of parametric traces." In: *Proceedings of the 6th Conference on Formal Methods in Software Engineering, FormaliSE 2018, collocated with ICSE 2018, Gothenburg, Sweden, June 2, 2018*. Ed. by Stefania Gnesi, Nico Plat, Paola Spoletini, and Patrizio Pelliccione. ACM, 2018, pp. 10–19. DOI: 10.1145/3193992.3193998.

[3] Ansem Ben Cheikh, Yoann Blein, Salim Chehida, Germán Vega, Yves Ledru, and Lydie du Bousquet. "An Environment for the ParTraP Trace Property Language (Tool Demonstration)." In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 437–446. DOI: 10.1007/978-3-030-03769-7_26.

[4] Yves Ledru, Yoann Blein, Lydie du Bousquet, Roland Groz, Arnaud Clere, and Fabrice Bertrand. "Requirements for a Trace Property Language for Medical Devices." In: *2018 IEEE/ACM International Workshop on Software Engineering in Healthcare Systems, SEHS@ICSE 2018, Gothenburg, Sweden, May 28, 2018*. ACM, 2018, pp. 30–33. URL: http://ieeexplore.ieee.org/document/8452638.

ACKNOWLEDGMENTS

First of all, I would like to thank the members of my PhD thesis jury for the time they have taken to examine my work, for their meaningful comments and questions, and for participating in the defense.

I would also like to thank my two PhD advisors, Yves Ledru and Lydie du Bousquet, for the remarkable amount of time they took to teach me the practice of research, and to help me improve my work. During this time, they have always been patient and kind, for which I am immensely grateful. They were also here to support me during the difficult yet rewarding experience that a PhD can be.

I also thank Arnaud Clère and Fabrice Bertrand for their availability, their helpful contributions, and for sharing their expertise.

A special thank to my close friends, Simon and Hugo, who were also PhD students at the time. Whether they were about our ongoing research, about our experiences as PhD students, or flat-out pedantry, our endless discussions were always a pleasure. I will miss them.

Finally, I deeply thank my mother who has always supported my decisions and believed in me accomplishing them.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ANR	*Agence Nationale de la Recherche*
DNF	Disjunctive Normal Form
EBNF	Extended Backus–Naur Form
FDA	Food and Drug Administration
FSM	Finite State Machine
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LGPL	GNU Lesser General Public License
LTL	Linear Temporal Logic
MCPS	Medical Cyber-Physical Systems
TRS	Term Rewriting System

1 INTRODUCTION

1.1 CONTEXT

A new generation of medical devices emerges to support increasingly more complex medical decisions and procedures. These Medical Cyber-Physical Systems (MCPS) combine data from novel sensors and existing modalities like scanners with elaborate software processing to assist caregivers, in the same way flight management systems help a pilot flying planes. For instance, Blue Ortho's MCPS allows performing Total Knee Arthroplasty more precisely (TKA, i.e. the placement of a femoral and tibial knee prosthesis), potentially dividing the number of revisions by two [67]. This is very beneficial to the patients because the replacement of a first prosthesis by another one incurs significant damage to the bones and very few patients can walk normally after a TKA revision.

Unfortunately, innovation and safety of such MCPS is hindered by the current software verification practices of the MCPS industry. The example of the Therac-25, a radiation therapy machine which killed or injured several patients, shows that bad development practices may lead to catastrophic situations [52]. As a result of this tragedy, national health agencies have reinforced their expectations on software development practices for medical devices, which are enforced with thorough audits conducted pre- and post-market. To enable the development of certifiable medical devices, several initiatives have promoted the use of *formal methods*.

Formal methods refer to mathematically based techniques for the specification, development and verification of software and hardware systems. The specifications used in formal methods are well-formed statements in a mathematical logic and the verifications are rigorous deductions in that logic. Successful deductions correspond to proofs that a system respects certain properties, such as its correctness with respect to its specification. Formal methods encompass highly diverse techniques ranging from fully automated, such as static type systems in programming languages, to completely manual, such as some of the most popular proof assistant.

Some formal methods have demonstrated their effectiveness for decades in the most safety-critical industries (e.g. defense, avionics [72], space [15], railways [33], or nuclear power plants [60]), but they failed to gain broader adoption. In the medical industry, the Pace-

maker Challenge [56] 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. However, full formal verification of new-generation MCPSs would be very difficult to achieve. This can be attributed to several factors:

- Caregivers use various combinations of medical devices and adapt procedures to fulfill the task at hand, making it difficult and costly to completely formalize, hence verify, the whole system a priori.
- Current regulations, such as ISO 62304 [44], do not ask for proofs of safety: pre-market validation is based on development process and risk analysis verifications; post-market surveillance can be limited to informal follow-ups. Nonetheless, some regulatory administrations like the United States Food and Drug Administration (FDA) are interested in evidences of safety, i.e. factual data on the actual device.
- Last but not least, a number of verification tools only target the C or Java programming languages, such as Frama-C [50] or Krakatoa [55], while the MCPS industry mainly uses C++ for its combination of abstraction and performance features (C being limited to embedded medical software and Java to interoperability with hospital information systems). Crocker gave a short overview of the existing approaches and difficulties faced for verifying C++ programs [22].

Most of the pioneering works for bringing formal methods to the MCPS field are performed by academic research groups, and significant efforts need to be done in order to transfer these formal approaches into industrial practice. The MODMED research project is one of those efforts, and this Ph.D. research was part of it. The project gathers a research laboratory, a medical device manufacturer and a software provided for medical devices, with the goal of improving evaluation of MCPS safety by introducing formal verification to the field. To maximize its relevance, MODMED research work is focused on the study of a worldwide used MCPS made available by one of the partners. Another key aspect of the MODMED approach is the use of so-called "lightweight" formal methods to introduce formal verification.

Since several years, a part of the formal methods community has promoted a "lightweight" approach to the use of formal methods. Based on the analysis of D. Jackson and J. Wing [45] which stated the prohibitive cost of full verification, the lightweight approach to formal methods advocates for:

- partial application of formal methods techniques, including specification of a subset of the system properties based on risk evaluation and cost-effectiveness,
- the choice of a language which allows automated tool support, or
- the use of tools that partially check the system but actually find bugs.

By trading completeness in favor of automation, lightweight techniques make the use of formal methods practical and within reach of a wider audience of software engineers. Runtime verification is an example of those techniques.

Roughly speaking, runtime verification is a set of theories, techniques and tools aiming towards efficient analysis of a system's executions and guaranteeing their correctness with respect to a specification. Practically, runtime verification consists in taking as input some system representation, performing some analysis, and yielding a verdict indicating the correctness of the system in addition to some form of feedback to the user. Runtime information must be directly obtained from the execution of the system and automatically analyzed. This approach perfectly matches the previous description of lightweight formal method as it only partially checks the system (a single execution at a time), but does not require a full model and can be completely automated.

1.2 THESIS STATEMENT

MCPSs are powerful systems that can be easily equipped with elaborate tracing facilities. Exploiting their execution traces and sensor data can provide us with an unbiased and precise understanding of their behavior in the field. Consequently, runtime verification seems particularly fit to the task at hand:

1. Systematic and automatic verification of execution traces can significantly improve the post-market surveillance of such systems, which is usually limited to ineffective and informal follow-ups of the device users. Trace analysis brings empirical evidence that the system exhibits the expected behavior. Moreover, it allows to understand how the system is actually used. For instance, it could allow validating safety-related assertions on users behavior [11] or hardware longevity [70].
2. It can complement the process-based validation of medical devices. Dedicated tools can use models as reference for assessing the functional coverage of existing tests (manual system tests as well as automated "white-box" unitary tests). This would contribute to establish the safety of the system by complementing

the test report reviews, based on code coverage, with an independent measure of the requirements captured in the model and covered by tests.

Recording execution traces is already a standard practice in most industrial software, but their effective exploitation requires adequate tools. A significant challenge is to provide medical software engineers with a simple formal language adapted to the expression of their most important requirements. In the field of runtime verification, several proposals — such as Dwyer's patterns [28] — allow expressing temporal properties at a higher level than temporal logic. These proposals are not directly suitable for the data-rich traces found in the medical domain.

1.3 CONTRIBUTIONS

In this thesis, we report on the analysis of a worldwide-used MCPS for total knee arthroplasty. After presenting the verification process for this device and highlighting its limitations, we explain how trace analysis can be used at several stages of development to overcome those limitations in Chapter 2. Next, we describe our process to identify requirements for a property language adapted to this MCPS in Chapter 3. After a brief presentation of runtime verification in Chapter 4, we show that existing temporal specification languages focus only on a specific subset of the identified requirements, and that a more practical solution can be designed in Chapter 5.

Based on the previous analysis, we propose ParTraP, a language dedicated to property specification for finite traces. Its design was driven by two objectives: allowing elegant specification of properties on traces produced by MCPSs, and being easy to apprehend by engineers with no training in formal methods. ParTraP uses a declarative style with a user-friendly syntax. It features intuitive temporal operators derived from the ones proposed by Dwyer et al. [28], with the difference that they can be composed for increased expressiveness. The language also features first-class support for data-carrying events with arbitrarily complex data layouts. ParTraP's semantics is fully formalized and implemented in a compiler packaged together with an Integrated Development Environment (IDE). It is available online under a free and open-source license. Chapter 6 shows an informal presentation of the language, Chapter 7 gives its formal semantics, and Chapter 8 describes its IDE companion.

We also propose a technique to evaluate the coverage of properties written in ParTraP over a set of traces in Chapter 9. Despite our efforts to keep ParTraP simple, correctly specifying temporal properties remains delicate. Complex properties may contain subtle cases, intricate to detect. Coverage information exposes them and the role

they played in the satisfaction of a property. In particular, errors in properties can be detected. We demonstrate this capability through several examples, and show that it can also help refining properties and detecting malformed traces. This coverage evaluation relies on the decomposition of properties into Disjunctive Normal Form, which is itself accomplished by a Term Rewriting System specifically designed for that purpose.

Chapter 10 concludes this thesis and discusses possible research directions and applications for ParTraP.

2

CONTEXT: AN INDUSTRIAL CASE STUDY

My Ph.D. research took place within the framework of the MODMED research project, funded by the *Agence Nationale de la Recherche* (ANR), the French national research agency. This project gathered three partners located in Grenoble, France:

1. MinMaxMedical, a company specialized in software for medical devices,
2. Blue Ortho, which develops medical devices, and
3. Laboratoire d'Informatique de Grenoble, a research laboratory.

Its main objective was to improve the evaluation of MCPSs safety and performance in the field by the mean of automated model-based verification of execution traces. The motivations behind this strategy were threefold. First, recording executions of MCPSs is relatively easy, thanks to their power and flexibility. Second, systematic and automatic verification of execution traces can significantly improve the post-market surveillance of such systems, which is usually limited to ineffective and informal follow-ups of the device users. Trace analysis brings empirical evidence that the system exhibits the expected behavior, and it allows understanding how the system is actually used. Finally, it can complement the process-based validation of medical devices and contribute to establish their safety before reaching the market.

To ensure the relevance of its research to the field, the MODMED project was centered around an industrial case study. The medical device under study was TKA, a surgery assistant designed and developed by Blue Ortho for Exactech, a US-based implant manufacturer. The TKA product was chosen for this project because, on the one hand, it appears as representative of new generation medical devices, and on the other hand, it has been used worldwide for several years. Blue Ortho already acquired more than 10 000 traces of execution resulting from real surgeries, which provide a rich raw material for the MODMED project.

In this chapter, we first present TKA in Section 2.1. Then, Section 2.2 briefly describes the international obligations for developing medical devices, and Section 2.3 details the methodology employed by Blue Ortho to meet those obligations. In Section 2.4 and Section 2.5, we describe the nature and role that execution traces currently play

in the life-cycle of TKA. Section 2.6 presents the potential benefits of trace verification for medical devices development, and how it fits in the product life-cycle. Section 2.7 concludes the chapter.

2.1 PRESENTATION OF TKA

Figure 2.1: Usage of the TKA product during a surgery (photography from Blue Ortho)

TKA is an application to guide total knee arthroplasty surgeries, i.e. the replacement of both tibial and femoral cartilages with implants. Figure 2.1 shows TKA in use during a surgery. The system is composed of software, mechanical and electronic components. The *station* on the left includes a touchscreen and a stereo vision camera on top of it. Beside the station, the system also includes a set of *trackers*. They are firmly attached to the bones of the patient and their position and orientation can be detected by the camera. The camera also detects a *pointer* device, which is used to acquire the position of some anatomical points. Two trackers and the pointer device can be seen in more details in Figure 2.2. The whole system can be manipulated directly by the surgeon or their assistants through the touch screen, and using the pointer device. Once the system has acquired all the necessary anatomic information, the surgeon can plan several cuts in order to place the prosthesis. The system will then help them to position cutting guides.

At a high level, performing a total knee arthroplasty with TKA mainly consists in the following sequence of operations during the surgery:

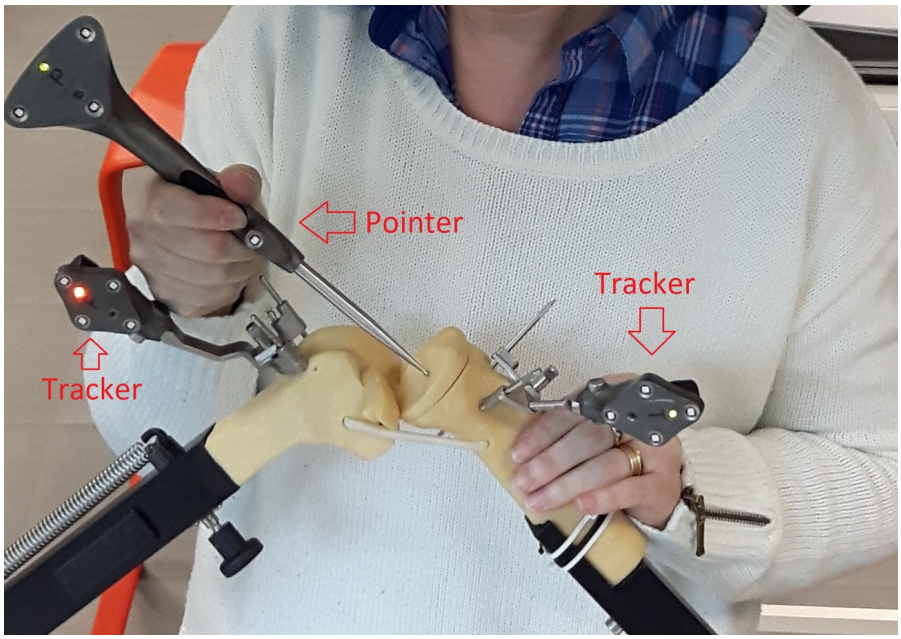


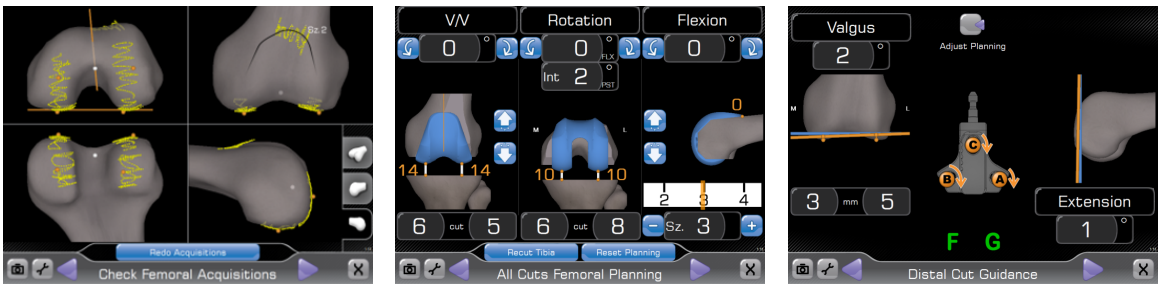
Figure 2.2: Pointer and trackers usage during the hands-on session

1. ACQUISITION After fixing the trackers to the patient's bones, the surgeon acquires a set of anatomic points by designating them with the pointer device. Other points require performing certain movements with the patient's leg, which are captured by the attached trackers. Once all the necessary points have been acquired, the system is able to construct a digital model of patient's anatomy. Several mechanisms and safeguards encoded in TKA software check that acquisitions are "correct" according to rules established by Blue Ortho. However, the software cannot judge whether the acquisitions correctly reflects the patient's anatomy. Therefore, surgeons are also trained to appreciate the quality of the acquisitions themselves. Figure 2.3a shows the interface of TKA allowing to visualize some of the performed acquisitions.

2. DECISION By combining the constructed model of the patient's anatomy and trackers position, TKA computes several metrics which are updated in live when the surgeon manipulates the patient. Thanks to this information, the surgeon is able to precisely adjust its preoperative planning for implant size and position. The interface displayed in Figure 2.3b allows the surgeon to set up and visualize their plan.

3. ACTION The surgeon installs some cutting guides on the patient's bones and adjusts them according to the information on screen. Thanks again to the live updating view, TKA directs the process so that the position of cutting guides matches the plan up to a degree or a millimeter. Figure 2.3c shows the interface corre-

sponding to this step. Once the guides are setup, the surgeon proceeds to cut the bones.



(a) Acquisition (b) Planning (c) Positioning

Figure 2.3: Screenshots of the TKA interface at different steps

Although this operation sequence appears very linear from the above description, the low-level order in which all the steps are performed is actually flexible and some of them may even be opted out. For instance, surgeons decide whether they want to acquire optional redundant anatomical points for additional confidence in the constructed digital model. The actual low-level ordering is computed at the beginning of a surgery according to surgeons' preferences and their needs for the present surgery. The order in which surgeons prefer to operate is called a *profile*, and is defined by a Blue Ortho representative during an interview. This feature introduces significant complexity in the software but it is a key for the adoption of the device by surgeons, who all have a preferred way to operate.

From a technical standpoint, TKA software is entirely written in C++ and composed of about 250 classes. It runs on machines using a Microsoft Windows operating system, to which the camera and screen are connected.

2.2 VERIFICATION AND VALIDATION CONSTRAINTS FOR MEDICAL DEVICES

Medical devices and their software are regulated by several international standards. Their development must follows a classical V-Model methodology complying with ISO 13485 [42], complemented with:

- an end-to-end requirements management complying with ISO 62304 [44], and
- a risk analysis of the medical device complying with ISO 14971 [43].

Before reaching clinics or hospitals in a country, medical devices must get certified by national or international drug agencies. The developers of the medical device have to demonstrate that the development process and the product are compliant with the aforementioned stan-

dards. Most notably, they must justify the results of the risk analysis for each software component, and show that the test coverage of each component is appropriate for its established risk level. This audit process is quite thorough and usually lasts several months. Anecdotally, Blue Ortho reported that reviewers' expectancies are high and that every objection or question they have must be answered with hard facts. Once the device has been certified and is used in production, the device is audited again periodically, as well as for minor revisions.

Medical device manufacturers are also expected to monitor all their devices in use, known as "post-market surveillance". However, this requirement is vague and follow-ups are usually limited to informal follow-ups of the device users, often ineffective.

2.3 BLUE ORTHO METHODOLOGY FOR TKA DEVELOPMENT

Blue Ortho followed a thorough methodology in the development of TKA, with a focus on quality beyond of what is expected from standard regulations. This methodology is based on a V-Model whose process is tweaked for the medical device industry. It is represented by the diagram in Figure 2.4, which was extracted from Blue Ortho Quality Management System. The Blue Ortho V-Model is split in 3

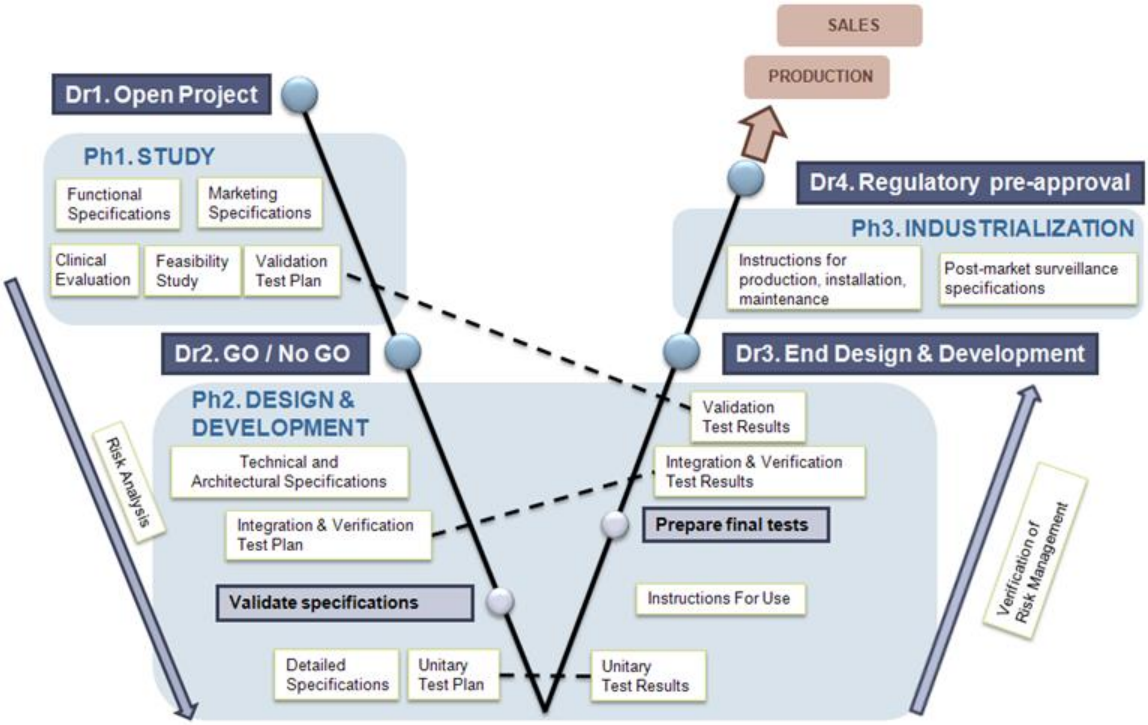


Figure 2.4: Blue Ortho's V-Model for TKA

levels depicted by dashed lines. Each of them is detailed by one or several confidential documents. As they are audited by drug agencies, these documents are high quality references which describe precisely

the design and implementation of TKA's software. From the least to the most detailed, we can name these 3 levels: User, Technical and Implementation.

2.3.1 *User Level: From Intended Use to Validation*

At the top-level, the design team, the development team and the quality managers along with some end-users (surgeons) write a document called "Marketing Specifications", describing the intended use of the product and the corresponding high-level user needs, along with a "Functional Specifications" document describing its components and high-level usability requirements. At the very end of the design and development procedure, the validation of the product will depend on the result of "validation tests" typically performed by surgeons on cadavers using a successfully verified device.

2.3.2 *Technical Level: From Design to Verification*

At the middle level, the design and development team together with the help of their engineering team write the "Technical Specifications". It is a confidential document describing the requirements on its software, and more specifically by detailing its technical functions, verification test plan, architectural design, interfaces, integration test plan, risk measures test plan, and technical traceability. Many of those are described through test scenarios and the "verification" of the product consists in performing all these scenarios manually and analyzing their results. "Technical Specifications" for TKA are 125 page-long and performing all verification tests, as required for major revisions of the product, takes approximately 1 person-month.

This document is more frequently named by other medical devices manufacturers "Sofware Requirements Specification" to follow the United States FDA terminology.

The whole TKA software is described through 32 groups named "System Functions". These 32 functions are further decomposed in a total of 319 test scenarios. They describe what is required to happen after each step, or at the end of a scenario. On average, there are 10 tests per function. Only 5 System Functions have more than 12 tests, and the most complex one has 46 tests. The effort necessary to test each case may greatly vary, depending on the number of initial environments.

These tests focus on the system accuracy and the user workflow. Accuracy cannot be tested with the medical device alone and is specifically tested with a specifically designed hardware test bench. Most system functions also include a test case where the tester is free to take unspecified user actions and to appreciate whether the resulting system behavior is normal. This illustrates the compromise between specifying tests with well-defined acceptance criteria (even informally) and the desire to verify the medical device under more situations without having time to precisely describe their inputs and outputs.

2.3.3 *Implementation Level: From Development to Unit Testing*

At the lowest level, engineers write "Detailed Specifications" documents for each software component in the form of unitary test plans. They implement the components and write unit tests for each component's function. The nature of those components is very diverse, ranging from geometric computation to graphical display. For instance, the graphical component in Figure 2.5 is responsible for accurately displaying named numerical values with the appropriate unit in various languages, and tested accordingly. TKA software contains 250 C++ classes which source code also contains the unit tests. The ratio "tests size over implementation size" can reach up to 3 depending on components. The test suite comprises more than 600 unit tests which are run automatically during several steps of the development.

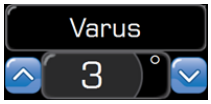


Figure 2.5: A TKA graphical component for numerical values display

Unit test results are collected and verified manually before performing the product verification. Being able to run these tests and obtain repeatable test results allows Blue Ortho to manage the proliferation of user options (about 30 different screens and 60 options), and translations (English, French, Spanish, German, Italian, and later Japanese, Korean, etc.). However, the limit of testing is well understood by Blue Ortho that takes test results as no more than "an evidence that the software performed well at least once". Consequently, they put a lot of efforts in software design to improve its quality, like statically tracking the many coordinate systems of geometrical primitives (anatomical axis and planes, physical instrument points, etc.) using dedicated types.

2.4 STATES AND TRACES

2.4.1 *Hierarchical Finite State Machine*

A notable example of Blue Ortho emphasis on design resulted in putting the responsibility to adapt TKA workflow to the surgeon's profile into a hierarchical finite state machine which models this workflow and drives the whole user interface (from physical tools to graphical widgets). This state machine is dynamically built according to the surgeon's profile. The instantiated state machine is logged at the beginning of a TKA execution. Figure 2.6 shows a short extract of a possible instantiation of the state machine, with parent states in blue and child states in white. In the following, whenever we talk about the "state" of TKA, we refer to the current state of the instantiated finite state machine.

2.4.2 *Execution Traces*

To monitor how TKA devices are used on the market, Blue Ortho equipped them with a mechanism to record execution traces, which

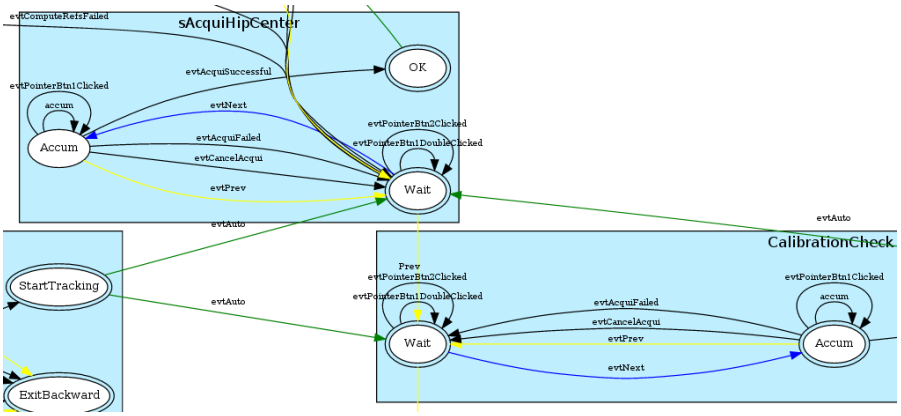


Figure 2.6: Short extract of a possible instantiation of the TKA state machine

are produced for each surgery. This mechanism was initially introduced to store sufficient information to understand the course of a surgery and possibly to identify failures. A trace reflects both the workflow of TKA software and of the surgery itself through a sequence of events. The nature of these events is very diverse and includes communication with the sensors, interactions with the surgeon, progress through the steps of the surgery, or complex computation results.

TKA traces are not particularly long: they are composed of about 3000 events on average. However, they are quite rich in data. Most of the events carry additional values, ranging from atomic values such as strings or numbers, to compound data such as 3D points or matrices. For instance, whenever the system acquires a cloud of points designated by the surgeon through the pointing device, it produces a single event containing that whole set of points (about 400). Figure 2.7 shows an excerpt of such an execution trace recorded by TKA during a surgery, with interesting information highlighted. We can see that each event is time-stamped and that the aforementioned state machine transitions are traced. A particular attention was paid to trace user input and sensors events because Blue Ortho wanted to be able to understand how the product would be used in the market. For instance, the last line contains the coordinates of the first point of a point cloud being accumulated.

Execution traces are also complemented with technical data such as screenshots of the graphical user interface taken at each step of the surgery, or the three-dimensional scene constructed from anatomic acquisitions. However, the traces contain little information about the internal values of the program (variables) and its control flow (method calls). Also, raw data acquired by the sensors is completely absent. One of the objectives of the MODMED project is to provide the means to easily trace such information in future products, or major revisions of TKA.

```
MSG 2015.11.11-02:18:01.145 | OPEN
MSG 2015.11.11-02:18:01.145 | Version : 1.15.3
...
MSG 2015.11.11-02:18:01.395 | [EventHandler::performStateEntry] Entering state : mainCasp
MSG 2015.11.11-02:18:01.395 | [EventHandler::performStateEntry] Entering state : mainCasp.
    Welcome
...
MSG 2015.11.11-02:18:02.659 | [EventHandler::performStateExit] Exiting state : mainCasp.
    Welcome
MSG 2015.11.11-02:18:02.846 | [EventHandler::performStateEntry] Entering state : mainCasp.
    Enter Patient Info
MSG 2015.11.11-02:18:11.207 | [BlueApp] Click on Btn Left at pos = (475,95)
...
MSG 2015.11.11-02:18:37.462 | [MainBlueWidget] Screen Btn Next clicked
...
MSG 2015.11.11-02:19:02.410 | [Profile::loadFromXML] file 'C:/.../Dr. XXX - ALL CUTS w ACB.
    bprofile' loaded successfully
...
MSG 2015.11.11-02:19:07.340 | [FoxDriver::connect] Device 166ec0dd01 connected
...
MSG 2015.11.11-02:19:43.859 | Marker Detected : 16aa1a9401, P001002
...
MSG 2015.11.11-02:20:37.913 | [EventHandler::performStateExit] Exiting state : mainCasp.
    CalibrationCheck.Wait
MSG 2015.11.11-02:20:37.913 | [EventHandler::performStateEntry] Entering state : mainCasp.
    CalibrationCheck.Accum
MSG 2015.11.11-02:20:37.929 | [  0] 0.0041657031046522519 64.562875356938733
    11.860463388262414
```

Figure 2.7: Excerpt of an execution trace recorded during a surgery with TKA

Blue Ortho collects the traces of all surgeries conducted with TKA. The product is being used worldwide for several years and more than 10 000 surgery traces have been collected. Currently, those traces are analyzed manually, but their increasingly large number is making that task less and less reliable.

2.5 POST-MARKET SURVEILLANCE

Once a medical device is certified for some market and actually used, Blue Ortho considers that analyzing the device usage in real surgeries is critical to ensure patient safety. For instance, they presented us a few studies on TKA that were done thanks to the collected traces, but also examples of studies that could not be completed by lack of tools or methods.

2.5.1 *Misuse Surveillance*

As a general usability principle, Blue Ortho chooses to only block the surgeon when the action would undoubtedly have dire consequences for the patient. Consequently, it is important to study how the TKA is used in the market to detect misuses. This allows warning users

about potential problems and advising them on how to avoid these problems in next surgeries. On the other hand, it may denote usability problems that should be tackled by Blue Ortho. In any case, Blue Ortho feels this is a very relevant activity to improve TKA safety and effectiveness.

A typical example is to verify that TKA is used within intended operating temperature range because it affects the camera accuracy. TKA software checks this prerequisite environment condition and the surgeon is warned about possible accuracy problems, but he is left responsible for using it or waiting for camera warm-up. The same requirement is checked on surgery reports that the surgeon can consult on a dedicated website. This is a successful example of using traces to educate users without taking on research and development resources. Unfortunately, this is an exception and most basic misuses are not detected resulting in user frustration, and involvement of rare and expensive research and development resources to diagnose trivial problems. For instance, although Blue Ortho stresses the fact that the camera must be placed next to the opposed leg of the one being operated, the company had to deal with situations where this requirement was not respected.

Blue Ortho also studied whether it was possible to detect the use of a leg holder based on knee and hip center distances. A leg holder is a tool to immobilize the thigh of the operated leg. Its use is incompatible with TKA which requires performing ample movements. Another study was made on 1000 traces to detect situations where some trackers are not properly fixed onto the bones. The problem with these studies is that it is difficult to establish a threshold on hip center gesture amplitude or point clouds metrics, etc. and Blue Ortho feels like they lack tools to perform more studies and implement more checks on surgery traces.

2.5.2 *Usage Studies*

Finally, an important outcome of surgery reports is to give users and manufacturers a feedback on TKA usage. On one hand, surgeons are given access to some anatomical metrics about their surgeries (before and after). On the other hand, the manufacturer can analyze which components are actually used to optimise the set of manufactured instruments. Research and development teams get information on the deployment of new software and hardware versions, and they can study how this affects surgeries time. For instance, Blue Ortho designed a second version of cutting guides for which they observed improvement in the time to position them.

2.6 THE NEED FOR AUTOMATED TRACE ANALYSIS

2.6.1 *Why Trace Analysis*

During its life cycle, a medical device is tested and analyzed in several contexts: development, qualification, manufacturing and exploitation.

1. THE DEVELOPMENT CONTEXT It corresponds to all activities that will create or modify the software or the system. They correspond to the initial development, but also to corrections during the maintenance phase and evolutions of the system. In the development context, traces will be produced during tests. They can be used to evaluate the correctness of the system by ensuring that software requirements are always satisfied.

2. THE QUALIFICATION CONTEXT It comes after development activities. It is aimed at demonstrating the correctness of the system and validating assumptions on its environment. The qualification phase also involves "acceptance tests" typically performed by surgeons on corpses using a successfully verified medical device. During these acceptance tests, one can expect that most of the requirements are satisfied as the system passed the development stage; thus, the focus will be on checking that the execution environment behaves as assumed, and that the product is used as foreseen.

3. THE MANUFACTURING CONTEXT Since TKA is composed of software and hardware, it involves a manufacturing phase where the system is manufactured and tested. Here the tests check the hardware for manufacturing defects. At this stage, the focus is on checking that requirements are satisfied by every manufactured device. While in the development and qualification contexts failure could result from software defects, in the manufacturing phase, the software should be correct and failures only reveal hardware defects, or incorrect execution of the tests by the tester.

4. THE EXPLOITATION CONTEXT It corresponds to the operation of a qualified system during a real surgery. The main activities in this context are the already mentioned post-market surveillance and usage studies. In this context, requirements should obviously not fail. Checking this on traces of numerous surgeries brings additional evidence of the quality of the qualified system. Similarly to the qualification context, the most relevant checks in the exploitation context are to verify that assumptions on the execution environment were realistic, and that the product is used appropriately in real conditions. Checking the assumptions on the traces of real surgeries helps to detect cases where the environment of the system is not adequate. Detecting

such traces may bring explanations on why something did not proceed smoothly, or require for more robustness of the system against the failure of these properties.

To summarize, Blue Ortho carries out three types of analyzes on TKA executions throughout its life-cycle: requirement verification, assumptions validation, and usage studies.

Except for unit testing during the development context, the first two are carried out completely manually. Obviously, this is tedious and error-prone. Usage studies are partly automated: the simplest ones are scripted and executed systematically on collected execution traces, whereas more complex studies require developing ad-hoc programs. Most, if not all, of those analyzes would benefit from being replaced – or complemented – by a more reliable and automated trace verification framework.

2.6.2 *Limitations of the Current Approaches*

Blue Ortho already automated some analyses on traces. They are currently using two different approaches:

1. Simple analyses are run against "surgery reports", which are hierarchically structured syntheses of TKA executions. Basically, those analyses are encoded as shell scripts looking for patterns in a report with XPath queries (i.e. XML queries). This approach is limited to simple analyses because surgery reports contain little information.

2. More complex analyses, such as usage studies, are run against the unstructured "technical logs". They are programs written in Python or C++ that encode properties of interest weaved together with extraction of relevant parts from the logs. They are not resilient to changes in the log format, and they are often developed and used with a throwaway state of mind.

Besides their respective drawbacks, they are also inscrutable by engineers or auditors not qualified with the in-house development techniques.

The drawbacks of both approaches can be attributed to two factors:

1. The lack of structure in traces limits the set of tasks that can be automated easily, and makes them not resilient to changes.

2. Analyses are encoded as ad-hoc imperative programs mixing together the logic behind the analysis and programming related tasks. Although scripts ran against surgery reports are using declarative XPath queries, they are still imperative programs. Furthermore, XPath, which was not designed to extract infor-

mation out of temporal sequences of events (traces), does not exactly reflect what the analysis is about.

2.7 TOWARDS AUTOMATED TRACE VERIFICATION

The first issue with the current approach of Blue Ortho for trace analysis, namely the fact that traces are not structured, can obviously be solved by using a structured tracing library. The second issue, i.e. having to write ad-hoc programs for each analysis, can be tackled by using a language specialized to this task. The two following subsections discuss those solution respectively.

2.7.1 *A Structured Tracing Library*

To produce structured traces, it is necessary to 1. decide on a structure and a format to represent it, and 2. use tracing instructions provided by a library supporting the chosen structure/representation. The part of the MODMED project dedicated to these tasks was handled by MinMaxMedical, and is not detailed in this thesis. In short, they designed a trace structure, several corresponding representation formats and a C++ tracing library such that:

- Tracing is efficient;
- Trace points leverage a maximum of static information from the program to automatically enrich events;
- The different representation formats are isomorphic; and
- The library provides the means to easily capture arbitrary data.

More details are available in the associated deliverable for the MODMED project [20].

2.7.2 *A Language for Trace Verification*

Writing ad-hoc programs to verify traces is tedious and repetitive, and consequently error prone. Resulting programs are also difficult to read and maintain as the property of interest is encoded in an operational form. On the contrary, a language dedicated to trace verification can offer abstraction from the underlying trace structure and to focus solely on the property intent. It can also significantly increase both readability and conciseness of specifications.

The design space for trace verification languages is vast and includes a number of trade-offs. This thesis is dedicated to finding the right one in the context of verification of modern medical devices. By "right one", I mean a language that would both fit its use-case, i.e. trace verification in the context of modern medical devices, and be suitable for industry adoption, especially for engineers without training in formal methods.

Finding this language requires to precisely understand which types of properties are to be verified. Chapter 3 reports on the analysis of properties of interest for TKA. Then, I compare several existing languages according to the results of the previous analysis in Chapter 4. Finally, we propose a formal language better suited to our needs in Chapter 6, and its associated toolset in Chapter 8.

3

REQUIREMENTS ANALYSIS FOR TKA

In this chapter, I present the methodology employed to identify properties of interest on TKA, and a comparative analysis of their most important features.

3.1 STUDIED REQUIREMENTS

The design documents of TKA were thoroughly analysed to identify the product requirements that could be verified, and 5 execution traces were studied to check whether the corresponding trace properties could actually be verified. They were carefully selected by Blue Ortho as illustrative of the "expected" workflow or unusual behaviors. Moreover, in order to target MODMED tools to goals that are deemed important by users in the industry and facilitate their adoption, Blue Ortho was asked to express whether the discussed properties were interesting from the industrial standpoint.

3.1.1 *Inappropriate Sources of Requirements*

USER LEVEL REQUIREMENTS The requirements listed in the "Marketing Specifications" and "Functional Specifications" are so general that they were not deemed interesting for the MODMED project. Also, the Validation activity performed by end users (surgeons) seemed too focused on the intended use to present challenging cases not already tested by the Verification activity. We will see below that both the Verification and Post-Market Surveillance are activities triggering user level requirements that are more interesting for the MODMED project.

WORKFLOW REQUIREMENTS Blue Ortho invested a lot of efforts to implement the desired surgery workflow as a hierarchical state machine driving the whole Graphical User Interface (GUI). This state machine was studied in detail as it looked like a good source for trace properties and its size was unusually large for controlling a GUI. Some properties were formalized and verified using prototypes, but the result was usually showing problems with the formalization rather than problems with the software. For instance, a property stating that "*the user should spend at least 5 seconds in each state*" has the purpose of checking that the user does not miss important steps by clicking too quickly. The property is however too broad because in the traces, two types of states have to be considered: states exposed to the user, often corresponding to clinical operations, and technical

states performing internal computations that are not exposed to the user. Many of those technical states are exited quickly by the system, falsifying the above property.

Moreover, the state machine is dynamically changed by TKA software to take into account user interactions like "camera reconnection" or "redo acquisitions" that are not in the normal workflow and these changes are not totally traced in the current version (in spite of Blue Ortho goal to trace all state machine changes). Finally, the transition conditions are only visible in the source code (making the state machine appearing as non-deterministic when considered alone) and duplicating them in properties represents a lot of manual work.

Blue Ortho's conclusion was that using the state machine to derive formal properties of traces would be a duplication of the effort already spent on design with a low probability of detecting real problems. MODMED partners were very sensitive to this appreciation since a major challenge of the project is the adoption of its tools by the industry.

On the other hand, Blue Ortho wanted to verify that:

1. general properties of the state machine implementation or specification (e.g., no other sinks than the end of the surgery), and
2. the state machine instantiated by TKA actually respects the surgeon's profile and general rules like: "*dynamically added workflows start and end in the same state of the normal workflow*".

GUI REQUIREMENTS Ensuring the GUI displays accurate and timely information to the surgeon is an important requirement. Unfortunately, current traces are somewhat limited in this area. User actions are traced but reactions of the GUI like audio feedback are not. A few screenshots are taken at each workflow step but the timely updates of GUI components are not (imagine the GUI freezes while the surgeon is adjusting cutting guide screws and cuts bones with the illusion that the guide is correctly positioned).

It would be important to be able to extract information from screenshots to automate the verification of some explicit or implicit requirements, such as the fact that text must be displayed correctly in all languages or that anatomical orientations of schemas are correct. However image processing was deemed out of scope of the project. Instead, it was decided to first tackle this problem by providing facilities in the trace library to trace GUI behavior without the risk of delaying GUI updates. Certainly, it cannot fully replace image analysis and will not detect problems in system GUI components.

ACCURACY REQUIREMENTS During our discussions, we realized that accuracy requirements like "*The precision of the computed hip center*

is less than 1 mm*"* would not be verifiable on execution traces as the absolute precision requirement requires ad-hoc test benches. What can be verified using only information coming from the traces is a weaker version of this requirement: *"All computed hip centers are located in a* 1 mm *sphere"* which tells something on TKA (inconsistent measurements probably indicate a misuse or a failure of TKA) but nothing on the ground truth (the sphere may be located at the wrong position).

REAL TIME REQUIREMENTS In the TKA study, we encountered few requirements on the real (wall-clock) time, and they were usually soft real-time requirements, in the sense that the time constraints may occasionally be violated without harm to the patient. Only a few hard real time requirements were found in "Technical Specifications", such as *"The system detects the new tracker in less than 10 seconds"*, while other real time requirements remain elusive like *"Each click leads to an immediate change of GUI"* or *"The position of the pointer is displayed in real time over the scheme of the bone"*. The main reason for being elusive is the necessity to distinguish such "usability" requirements from critical performance requirements. In particular, drug agencies may unduly interpret the fact that trackers should be detected in less than 10 seconds as a critical performance requirement (whereas it is not) and investigate: how it was measured, how the design helps to fulfill it, etc.

UNIT TESTS Unit tests are a way to implement some aspects of a more general requirement, so we wondered whether they would represent a relevant source of requirements. However, Blue Ortho felt like the software design patterns they employed combined with unit tests are adequate to independently test software components and did not feel the need for using new tools at the implementation level (except tests coverage tools). In practice, Blue Ortho experienced many more problems with integrated off-the-shelf hardware components than homemade software components.

Nonetheless, it could be interesting to see if some unit tests can be rewritten as properties since they may be easier to write with a dedicated language, and they could be verified in conditions that the tester did not anticipate as problematic. This approach is not currently applicable to TKA because its execution traces focus on capturing what happened in the environment and the values of intermediate variables usually checked by unit tests are not traced. Replaying traces was examined as a way to use existing traces and Blue Ortho experimented it with a prototype. However, limitations quickly appeared, such that the absence of raw input data preventing the re-computation of some data. Thus, apart from a few examples, we

did not further study test plans, described in the "Detailed Specifications", nor unit test programs.

3.1.2 *Relevant and Representative Requirements*

TECHNICAL LEVEL REQUIREMENTS The requirements described in the "Technical Specifications" document were the most important source for deriving properties of interest on TKA. As a result, a list of 43 requirements in 11 "functions" described in the "Technical Specifications" was determined relevant based on the interest expressed by Blue Ortho and their adequacy to the challenges MODMED addresses. The 21 remaining functions were not considered because they were almost identical to the ones already considered. The high relevance of this document content can be explained by two factors.

First, Blue Ortho was looking for better tools to support the verification activity because:

1. it is still very labor-intensive,
2. it includes hardware components that are harder to test effectively, and
3. its accuracy relies on the tester experience.

Indeed, performing all the test scenarios and interpreting the results frequently require specific knowledge that is not, or cannot be fully described in the "Technical Specifications". For instance, the requirement that "*The virtual keyboard allows to fill the form as expected*" uses the expression "as expected" instead of fully describing how a keyboard works. The verification activity is fully described in the "Technical Specifications", hence the relevance of the document.

Second, the verification process is described through test scenarios to be performed by the tester. They are usually composed of a sequence of actions together with an expected result. This type of scenario can be precisely formalized as a temporal property, and automatically checked on execution traces.

USAGE STUDIES AND POST-MARKET SURVEILLANCE Another source for finding properties was the interviews with Blue Ortho engineers. They explained their process and findings when confronted to bizarre situations reported by surgeons, as well as several studies they conducted in order to improve the product. They often ended up writing ad-hoc programs looking for patterns in traces, from which we extracted properties. Although they are not requirements and they do not have to be satisfied by all execution traces, it was important to account for those properties in the study. Tracking the aforementioned issues is a considerable time expense for Blue Ortho, which could benefit from tools supporting this activity.

3.2 LIST OF REPRESENTATIVE PROPERTIES

Out of all the properties I gathered as relevant, I extracted a shorter list of 15 properties that were deemed representative, based on the temporal relationships featured between events, the type of event data, and the operations performed on event data. tka properties are now listed in arbitrary order.

Property 1. *The trace contains a step "redo acquisitions".*

The "redo acquisition" step allows the surgeon to correct his previous acquisition. It is not part of the standard procedure flow and, therefore, interesting to detect.

Property 2. *The temperature of the camera stays within a given interval.*

If used in proper conditions, the camera temperature should not deviate from the range where its precision is guaranteed.

Property 3. *The distance between pairs of hip centers is less than d.*

This property asserts that the algorithm computing the hip center is stable, i.e. gives similar results for consecutive acquisitions.

Property 4. *The distance between the hip center and the knee center is greater than d.*

A violation of this property could reveal an abnormal positioning of the patient or the sensors.

Property 5. *If the medial malleolus is farther from the camera than the lateral one, a warning is issued.*

A violation of this property may reveal that the 3D camera was installed on the wrong side of the patient.

Property 6. *The user never skips a screen.*

The surgeon is expected to spend sufficient time to appreciate the information showed on the display before going to the next screen.

Property 7. *The acquisition of a point succeeds if and only if the probe is stable.*

If the surgeon moves the probe tip during an acquisition, it should not be accepted.

Property 8. *The protocol "redo acquisitions" only proposes already performed acquisitions.*

The system should not offer the user to redo acquisitions that were never performed.

Property 9. *Detecting a new tracker produces a dialog asking for replacement confirmation.*

Property 10. *The state TrackersConnection is unreachable until the camera is connected.*

The system should not reach a state dependent on the camera until the camera is connected.

Property 11. *A replaced tracker is not used until it is registered again.*

Property 12. *The action "previous" cancels the current point cloud acquisition.*

Acquiring a cloud of points takes a few seconds and can be cancelled. In this case, the current acquisition should not succeed.

Property 13. *All the necessary trackers are seen before entering the state TrackersVisibCheck.*

To proceed, the system requires a set of trackers depending on the profile in use. All these trackers should be seen at least once before entering the state TrackersVisibCheck.

Property 14. *On the tracker connection screen, a tracker is shown if and only if it is necessary.*

Only required trackers are shown to the user.

Property 15. *In the state TrackersConnection, not detecting a single tracker for 2 minutes produces an error message.*

3.3 TYPES OF PROPERTIES AND THEIR USE

Trace properties are very versatile in the analysis of medical device executions. They can be used to encode the requirements of a system, but also to detect specific behaviors of the system. In this section, we propose a classification for trace properties for medical devices according to three categories: *required*, *assumed* and *usage* properties.

REQUIRED PROPERTIES Required properties must be ensured by the system, and more precisely by its software. tka properties correspond to requirements on the software. For instance, one of the requirements of the TKA software states that it should check the stability of the probe before validating the acquisition of a point. It can be restated as a required property: "*The acquisition of a point succeeds if and only if the probe is stable*".

Checking these properties on the traces should always succeed, otherwise it would reveal a failure of the software. Ideally, these properties should be formally proven. This is why we refer to these as *required* properties. In the medical device industry, a more pragmatic approach is to provide verified traces as evidences that they are fulfilled by the product, and not full proofs.

ASSUMED PROPERTIES Assumed properties should be ensured by the environment of a system. They appear as assumptions on the behavior of this environment. If the environment fails to fulfill these properties, the behavior of the system may be affected. In the context of TKA, the property *"The temperature of the camera stays within a given interval"* is an example of assumed property. If the temperature is outside this range, the precision of the camera may be affected.

Violations of assumed properties by the environment may or may not be detectable by the software. When the environment does not behave as expected, or is suspected not to, the desired response from the software is not obvious. In such a situation, TKA is designed to not stop assisting the surgery so that the surgeon remains in control, but it repetitively displays modal warnings.

Checking these properties on traces is expected to succeed because the surgeon and his team are expected to use the system in the prescribed conditions. If one of these properties is not satisfied, it may be an explanation for difficulties arising during the surgery and it does not necessarily reveal a defect of the system. tka assumptions on the behavior of the environment will be referred to as *assumed* properties.

Note that properties can be both required and assumed. This happens for properties whose satisfaction depends on the environment and the software implementation. For instance, consider the TKA property "*The distance between pairs of hip centers is less than d"*. It can result from the use of a leg holder, which violates assumptions on the environment, or from a wrong calculation which reveals a software failure. Since the trace does not record that a leg holder was used, a violation of this property leaves two possible causes.

USAGE PROPERTIES Some medical devices such as TKA may offer different workflows, depending on the precise nature of the intervention and on the surgeon's choices. Properties can be checked to understand how the system was used.

For example, this is the case of the property "*The trace contains a step 'redo acquisitions'"*. The "redo acquisition" step is triggered by the surgeon and may reveal that it is difficult to have all acquisitions right at the first attempt, or that the surgeon is not trained enough to use the system. Checking these properties helps understand the way a system is used, but does not reveal a particular failure of the system or its environment. It can be exploited to identify potential evolutions of the system (e.g. efforts should be done to facilitate acquisitions). This is why we refer to such properties as *usage* properties.

Statistics can be computed on the number of traces satisfying a given usage property. At longer term, "quantitative usage properties" might be considered, reporting quantities instead of Boolean values

PROPERTY	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Required			✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Assumed		✓	✓	✓		✓									
Usage	✓					✓									
No. of event types	1	1	1	2	3	2	2	2	2	2	3	3	4	2	4
Parametric		✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓
Temporal						✓	✓	✓	✓	✓	✓	✓	✓		✓
Restricted scope								✓	✓	✓	✓		✓		✓
Geometric predicate			✓	✓	✓	✓									
GUI predicate														✓	
Physical time						✓									✓

Table 3.1: Classification of the 15 selected properties

like a number of occurrences of an event or the duration of a step or the distance between anatomic points.

The upper part of Table 3.1 synthesizes the classification of the 15 selected properties according to the three aforementioned categories (required, assumed and usage properties). Required properties is the most populated class, which is representative of the TKA properties we gathered.

3.4 ANALYSIS

From the selected set of the 15 representative properties, we identified several important characteristics based on their relevance to the MODMED project and/or their frequency of occurrence. The characteristics that were deemed important are detailed in the following list:

- The number of different event types involved. This often correlates with the complexity of the property.
- Whether the property is *parametric*, i.e. if it constrains the data carried by certain events, possibly relatively between them.
- Whether the property is *temporal*, i.e. if it constrains the order of two or more occurrences of events.
- Whether the property applies to restricted scope, or interval, of the trace; this interval is delimited by given event occurrences.
- Whether the property relies on geometric computations on data extracted from event parameters.
- Whether the property uses predicates on the GUI (e.g., a given button should be disabled).

- Whether the property involves physical-time.

The lower part of Table 3.1 synthesizes the classification of the 15 properties presented in the previous section. In the following, I will detail this classification for a couple of properties.

Let us consider Property 3, stating that the distance between pairs of hip centers is inferior to a given threshold. It involves several events of a single type, reflecting the acquisition of a new hip center and that are parametrized with the acquired point. tka event parameters must satisfy a geometric constraint. A violation of this property could indicate that the patient was not installed as expected (e.g. the surgeon should not use a "leg holder" which locks the patient's leg) or, if the patient was correctly installed, that the algorithm computing the hip center is not stable.

Property 12 is a required property stating that the action "previous" cancels the ongoing points cloud acquisition. In other words, triggering the action "previous" during an acquisition prevents this acquisition from succeeding. This property involves three different event types, reflecting the action "previous", the beginning of an acquisition, and the success of an acquisition. No event parameter is needed, which makes the property non-parametric. However, it is temporal since the occurrence of the "acquisition success" event is constrained by other event occurrences.

The results of this classification are in accordance with our expectations: the properties are very diverse and rely heavily on data parameters. On the contrary, physical time is rarely involved in the identified properties despite our expectations. tka properties will guide our design of a trace property language adapted to the context of medical devices.

4

RUNTIME VERIFICATION

Runtime verification is a lightweight, yet rigorous, formal method that analyzes a single execution of a system. It can be used to ensure the reliability, safety, robustness and security of a system. Although runtime verification can be applied to any system whose behavior can be observed, we only consider software systems and their execution environment in this thesis.

Runtime verification is sometimes also referred to as *monitoring*, although the latter only suggests some form of observation over a period of time, whereas the former implies a notion of *correctness* with respect to some specification. Indeed, the essence of runtime verification is to check the correctness of the runtime behavior of a system.

Runtime verification can be applied before deployment of the system, for instance as part of the testing process, or after deployment of the system. The latter constitute its most unique distinguishing feature compared to other verification techniques: a system can be analyzed while operating in its production environment, where unexpected situations may occur.

In this chapter, we describe the three steps that compose the spine of runtime verification. First, we consider how to specify the valid behaviors of a system in Section 4.1. Next, Section 4.2 presents the different strategies to make a system observable. Finally, we describe some popular techniques and setups to verify that the system executions conform to its specification in Section 4.3.

4.1 SPECIFICATION OF THE SYSTEM BEHAVIOR

Generally, the term *behavior* is used to describe how a system changes or acts over time. Recall that we include both the software and its execution environment in the system. Therefore, the behavior can describe changes in the internal state of the software, the actions it performs, which may or may not affect the environment, and changes in the environment. For instance, Property 2 in Section 3.2 describes the temperature of the camera of the TKA system.

Of course, it is only useful to specify behaviors that can be observed. In this section we focus on the nature of those observations, whereas their practical realization is the topic of the next section. As many other works in the field of runtime verification applied to software

systems, we choose to observe the actions or state changes made by a system through discrete *events*. An alternate approach is to observe the system through a collection of signals, which are functions from time points to a value domain. This view is particularly useful when dealing with continuous time [54] and well suited to runtime verification targeted at hardware systems [46, 47], but less so for software systems, which are inherently discrete. The sequence of events observed during a single execution of a system, and therefore the system behavior for this particular execution, is called an *execution trace*, or simply a *trace* [64].

Generally, individual events carry any kind of information about the system behavior. They can be as simple as a name for something that happened. For instance, `CameraConnected` could be observed right after the camera is connected. Most recently, emphasis within the research community has been on so-called *parametric events* [1, 6, 24, 27, 38]. Such events carry data values of interest as *parameters*. For instance, the event `Temperature`(t), which has a single parameter t, could indicate the observation of a temperature sensor with a value of t. We will come back to the different ways to model event parameters in the next chapter. Regardless of the chosen model, the set of observable events, called *alphabet*, directly constrains the system behaviors that can be specified.

The desired behavior of a system is described with *properties* or *specifications*. In the field of runtime verification, it is common to differentiate the two:

- A *property* informally describes the desired behavior of a system. This behavior is described according to the observations we can make about the system, i.e. which traces are valid, and which ones are not. Abstractly, a property can be seen as the subset of execution traces satisfying the property out of all the possible traces. As we usually describe properties in natural language, they are often ambiguous.
- A *specification* is a formal encoding of a property in a given *specification language*. If the later is well-defined, any ambiguity shall be resolved during the specification process. The design space for specification languages is quite large and is presented in more details in the next chapter.

We will later refer to properties and specifications describing the behavior of a system as *temporal properties* and *temporal specification*, respectively.

Let us illustrate the difference between properties and specifications with an example. Consider a system which can perform some kind of action, reflected through the observation of an event a. This is the only observable event. In other terms, the alphabet is the singleton $\{a\}$.

Thus, an execution trace simply is a sequence of a, possibly empty. Then an example of property for this system could be "*the system only performs an odd number of actions*". It informally describes a subset of valid execution traces, namely the ones of odd length. We can give a formal specification for this property with the *regular expression*

$$a(aa)^*$$

whose language is the set of traces with an odd number of a.

4.2 SOFTWARE INSTRUMENTATION

As already mentioned, it is only useful to specify behaviors that can be observed. We now present some common strategies to actually observe a system execution.

The mechanism used to extract traces of events from a software during its execution is called *instrumentation*. Software instrumentation consists in adding extra code into a software component so that it produces a trace when executed. It is a well-established method employed in many applications, including runtime verification, and most heavily in software profiling. The nature of the traces and the types of analyses performed on them depends on the application. For instance, software profiling consists in gathering precise metrics about the execution of a software component in a system, such as its efficiency or memory consumption. In runtime verification, we mainly want to trace what the system is doing in order to verify correctness properties (although it is also possible to verify non-functional properties such as a certain level of availability).

Software instrumentation can be performed at two levels. In *source code* instrumentation, extra instructions are added to the software source files [13, 69]. The other approach is to perform instrumentation at the *binary* level [10, 51, 58]. Any type of executable code can be targeted, such as native machine code or bytecode. Although other combinations are possible, the classical approach is to instrument source code statically, i.e. at compile time, and to instrument binaries dynamically, i.e. at execution time.

Source code instrumentation is the preferred level in runtime verification. Indeed, it is easier to capture and to reason about the logic of a program at that level. Source code can be instrumented manually or automatically. The manual process of adding instructions that produce a trace reflecting the behavior of a program is a common practice in the software industry, known as *logging*. Popular logging frameworks include Log4j[1] for Java and Pantheios[2] for C++. This process is performed on an ad-hoc basis, according to the local criticality

1 https://logging.apache.org/log4j
2 http://www.pantheios.org/

perceived by the developer. The resulting trace is likely to contain highly relevant information, but its partiality prevents observing certain behaviors. On the contrary, automatic source code instrumentation is a systematic approach. A common technique is to use *aspect oriented programming* frameworks to automatically insert logging instructions. Such frameworks provide a mechanism to statically enhance a program with additional behaviors, without modifying the original source code. AspectJ [49] and AspectC++ [73] are two popular implementations for Java and C++, respectively. There also are implementations specialized to certain development and runtime environments, such as Android [30]. In runtime verification, aspect oriented programming is typically used to insert logging instructions at source code locations of interest, such as function calls. The set of locations that must be instrumented may be given manually, or derived automatically from a specification.

It is worth mentioning that code instrumentation introduces a computational and memory overhead, which may interfere with a system and perturb its behavior [34]. As a consequence, some defects may also disappear, and the verification of timing related properties may become unreliable. Controlling the instrumentation overhead is generally tackled with sampling-based techniques, but they introduce uncertainty in the monitoring result [14, 34, 37, 40].

4.3 VERIFYING EXECUTIONS

Runtime verification allows checking the conformance of an execution against a specification. The oracle performing this check is called a *monitor*. When enough information of a system's execution has been observed, a monitor may decide whether the specification is satisfied or violated by the execution. This decision is called a *verdict*.

In a runtime verification setup, monitors are considered part of the trusted computing base; therefore, they must be correct, i.e. for any execution, a monitor produces a verdict in accordance to the semantics of the specification language. For this reason, monitors are usually generated by automated synthesis procedures that take a syntactic representation of the property as input. Of course, this offsets the correctness issue of monitors to the synthesis procedure. However, if the procedure is correct, then all the generated monitors are also correct.

An important characteristic for a monitor is its operating mode: we distinguish *online* and *offline* modes. In *online* mode, the monitor runs concurrently to the monitored system and observes its execution step by step. In this case, the finite trace that is observed can be viewed as a prefix of the possibly infinite behavior of the system. In *offline* mode, we only consider terminated executions of a system, whose traces are

stored and processed as a whole by the monitor. While online analysis opens up the possibility to mitigate the violation of a specification when detected [32], offline analysis is much less intrusive.

Besides their ability to react to violations, online monitors are essential for systems running indefinitely. Indeed, offline monitors require a finite execution. It is worth mentioning that not all properties are interesting to check on infinite executions: there exist properties for which an online monitor could never yield a verdict. This occurs when any verdict decision taken by the monitor could be contradicted by a possible continuation of the current execution. For a monitor to remain impartial in such situations, i.e. never produce a verdict that could be contradicted in the future, the only solution is to never reach a verdict, effectively making the monitor useless. For instance, the property "*whenever* `a` *happens,* `b` *happens in the future*" is not worth monitoring. A property for which online runtime verification can produce impartial verdicts is said *monitorable*. The commonly used definition of monitorability (i.e. whether a property is monitorable) was proposed by Pnueli and Zaks [62].

Runtime verification has been active research topic for more than 15 years. Challenges in the field span many dimensions [31], of which we introduced a few in this chapter. One of the most important is perhaps the choice of a specification formalism. The design space is vast and this choice has direct implications on other dimensions, such as the performance of the generated monitors. It also has indirect implications. For instance, a formalism designed with end-users in mind is more likely to be adopted than a formalism that completely disregards usability. In the next chapter, we study some popular formalisms for temporal property specification.

5

TEMPORAL SPECIFICATION LANGUAGES

The objective of this chapter is to study the suitability of some popular temporal specification languages with respect to the requirements driven by the MODMED project, namely the specification of temporal properties relying on structured data carried by events with a formalism amenable to industry adoption.

5.1 OVERVIEW

5.1.1 *Pattern Systems*

An important challenge for the MODMED project was to make temporal specification accessible. A successful technique to assist users with temporal specification is to provide high-level temporal patterns with a verbose syntax. Dwyer et al. proposed a major step in that direction with a pattern system covering most of the temporal requirements of a very large study [28]. Those temporal patterns are domain agnostic and relatively easy to use. Within this system, each specification is composed of a *pattern* and a *scope*.

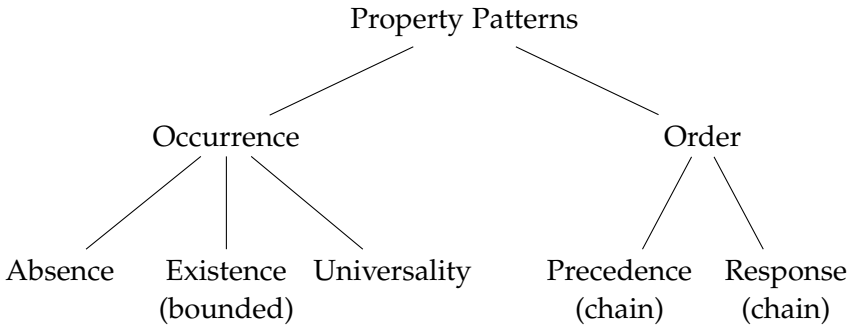


Figure 5.1: The 8 property patterns proposed by Dwyer et al.

PATTERNS Patterns constrain the presence or absence of events, or the order in which they may appear. The 8 patterns are listed as the leaves of the hierarchy in Figure 5.1. Without surprises, the Absence pattern forbids any occurrence of an event, the Existence pattern enforces at least one occurrence, and the Universality pattern requires for an event to constantly occur. The bounded variant of the Existence pattern allows specifying bounds on occurrences count of an event, such as "at least twice and at most five times". The Precedence pattern describes a relationship between a pair of events where the

occurrence of the first is a necessary pre-condition for an occurrence of the second. The Response pattern is the converse: an occurrence of the first event must be followed by an occurrence of the second. Finally, both order patterns have a "chain" variant used to express order relationships between sequences of events.

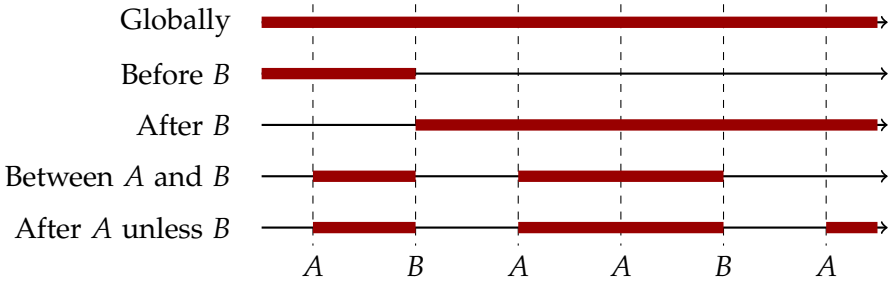


Figure 5.2: Graphical representation of the 5 scopes proposed by Dwyer et al.

SCOPES Patterns are paired with scopes, which describe time intervals where a pattern must hold. They are delimited by event occurrences, as depicted by colored intervals in Figure 5.2 for each of the 5 scopes. Although their meaning should be clear from the figure, a couple of points are worth mentioning:

- all intervals are left-closed and right-open, i.e. the event starting the scope is included whereas the one ending it is not;
- the Before and After scopes always refer to the first occurrence of the designated event;
- intervals in the Between/and scope are always terminated by the first occurrence of the right-hand side event;
- the After/unless scope is usually referred to as the "weak" variant of Between/and since the right-hand side event does not have to occur, in which case the last interval is only half-bounded, as illustrated in the figure. This distinction must not be neglected during specification.

EXAMPLE The specification process for a given requirement consists in identifying the events of interest, the temporal pattern and the scope where it should hold. Let us consider an example. We want to specify the following requirement: "*after trying to open a connection, a network error should produce an error message*". The temporal pattern behind this requirement is a Response: the occurrence of a network error should lead to the display of an error message. Moreover, this response must only hold after trying to open a connection, which corresponds to the After scope. Let us assume that trying to open a connection, detecting a network error, and displaying an error mes-

sage are reflected by the events OpenConnection, NetworkError, and ErrorMessage, respectively. Then, we could write the property as

> ErrorMessage responds to NetworkError after OpenConnection.

Dwyer et al. did not propose a formal syntax for their pattern system, which is why we prefer not to call it a "specification language". Examples of concrete syntax can be found in works that adopted this system, such as the Bandera Specification Language [21]. Nonetheless, Dwyer et al. provided a formal semantics for patterns and scopes based on their intuitive natural semantics. It was defined by giving a translation of each possible pair of pattern and scope into several logics.

OTHER PATTERN SYSTEMS Because the intuitive natural semantics of patterns is quite subjective, their meaning may not be obvious based on their names alone, especially regarding corner cases. For instance, the Response pattern does not require for the left-hand side event to occur, in which case the pattern is vacuously true. While this definition may make sense to some, it may be surprising to others. Smith et al. proposed to refine the pattern system of Dwyer et al. with additional mandatory clauses in PROPEL [71]. Its guided interface ensures that all subtleties are lifted during specification, such as whether one should use the strong or weak variant of the Between/and pattern.

Other patterns systems and pattern-based languages have been proposed, tailored to different needs. For instance, The Requirement Specification Language is a pattern system with limited temporal relations but a high emphasis on real-time [25], which is completely absent in the work of Dwyer et al.

While being relatively easy to use, pattern systems have a severely limited expressiveness due to the finite number of possible combinations. The SALT language [9] addresses that issue with *composable* patterns and common temporal operators behind a verbose syntax. It features many constructs, ranging from timed temporal operators to star-free regular expressions.

All of the aforementioned pattern systems and pattern-based languages were designed with finite-state verification in mind. As a consequence, they do not support parametric events.

5.1.2 *Linear Temporal Logics*

Pnueli introduced the Linear Temporal Logic (LTL) in 1977 for the purpose of formal verification of computer programs [61]. It has become a reference for temporal property specification and it is the most commonly used temporal logic in runtime verification. LTL can

be described as propositional logic augmented with temporal operators allowing to specify what should, or should not, happen in the future.

Given a set of *atomic propositions* AP, the set of LTL formulas over AP is defined inductively as follows:

- *true* and *false* are LTL formulas;
- if $p \in AP$, then p is an LTL formula;
- if φ and ψ are LTL formulas, then $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \implies \psi$, $\mathbf{X}\varphi$, $\mathbf{G}\varphi$, $\mathbf{F}\varphi$ and $\varphi\mathbf{U}\psi$ are LTL formulas.

The operators **X**, **F**, **G** and **U** are the temporal ones, while the others are the same as in propositional logic. Informally, $\mathbf{X}\varphi$ means that φ should hold in the next state, $\mathbf{F}\varphi$ that φ should hold at some point in the future, and $\mathbf{G}\varphi$ that φ should always hold, i.e. in the current state and in the future. Finally, **U** stands for "until" and $\varphi\mathbf{U}\psi$ means that φ should hold until ψ becomes true. For instance, the LTL formula

$$\mathbf{G}(\text{OpenConnection} \implies (\mathbf{G}\ \text{NetworkError} \implies \mathbf{F}\ \text{ErrorMessage}))$$

encodes the property that after trying to open a connection, a network error should produce an error message.

The original LTL was defined with model-checking in mind, and its semantics is not suitable for runtime verification. The issue lies in the fact that runtime verification is concerned with finite traces (or finite prefixes of infinite executions). Bauer et al. offer a formal treatment on adapting LTL for runtime verification, LTL_f, as well as its timed variant, TLTL_f, and discuss different trade-offs and their implications [7, 8]. The Counting Fluent Temporal Logic (CFLTL) is another interesting adaptation of LTL to event-based traces [66]. It features a simple but powerful mean of counting event occurrences and comparing them.

In event-based runtime verification, atomic propositions in LTL formulas correspond to event names, and they are evaluated to true whenever the given event occurs. While convenient, this view prevents using event parameters, thus is not suited to parametric runtime verification. To deal with parametric traces, many first order temporal logics have been proposed, most of them also deriving from LTL. Stolz introduced free variables and quantification in next-free LTL with parametrized propositions [74]. FO-LTL$^+$ [36] is another approach adding quantification to LTL. It targets data-rich XML traces and events parameters may exhibit a hierarchical structure. Parameters can be accessed through quantifiers over a domain described with XPath queries. In spite of the additional capabilities, FO-LTL$^+$ retains the simplicity and conciseness of LTL. However, the only way to

access events is through quantification over their parameters. In consequence, single-valued parameters must also be quantified, which makes formulas harder to understand. Basin et al. proposed a semantics and an algorithm for the Metric First Order Temporal Logic (MFOTL) [6] and implemented it with the MONPOLY tool [16]. They further extended MFOTL with SQL-like aggregation operators [5]. EAGLE [2] is a radically different and powerful temporal logic as it does not derive from LTL, but still encompasses it. This very succinct logic also features parametric events and data quantifiers.

5.1.3 *Finite State Machines*

Finite State Machines (FSMs) form another popular specification formalism in runtime verification. Transitions are labeled with event descriptions and taken whenever a matching event is observed. Whereas pattern systems and temporal logics are declarative, FSMs encode specifications in an operational style.

FSMs are especially popular in the so-called *parametric trace slicing* framework [18]. In this framework, events are composed of a name and a tuple of parameters, and properties can be quantified over those parameters, but only at the top level. In First-Order Logic, this restriction would result in formulas having quantifiers only at very left of the formula and spanning it entirely. A direct consequence is that the entirety of a property can be instantiated for a particular valuation of the quantified parameters, which can correspond to a particular FSM for this valuation. For instance, FSMs constitute the core of the expressive Quantified Event Automata [1] and its efficient implementation MARQ [63]. Mufin [26] also uses parametric trace slicing and relies exclusively on FSMs. It is faster but less expressive than MARQ.

5.1.4 *Rule-Based Systems*

The use of rule-based systems for runtime verification has also received some interests. This model seems well suited for processing data rich events, as exemplified by the RULER system [4] or the more efficient Logfire [38]. Rules take the form

$$\text{condition}_1, \ldots, \text{condition}_n \implies \text{action},$$

where the conditions depend on a memory of facts and the action can add or remove facts, or yield a verdict. Constantly managing a database of facts through action rules makes rule-based system highly operational.

5.1.5 *Hybrid*

Some formalisms or monitoring tools support several specification styles.

LogScope [3] is a solution proposed in a context similar to MODMED's one. It targets offline verification for data-rich traces. LogScope is composed of a high-level pattern language that compiles down to FSMs, which can also be used to encode more complex properties. The high-level language is similar to a pattern system, with the differences that it has a single scope (equivalent to the Before scope from Dwyer et al.'s system) and versatile constructs to specify event sequences. This kind of two-levels systems have the advantage of providing a very simple high-level language. However, they also force the user to learn two formalisms, including a low-level one, often with different paradigms (e.g. declarative and operational for LogScope).

JavaMOP [48] also supports several formalisms, called "logics". It is another incarnation of parametric trace slicing, being famous for making it efficient. Contrary to the two-level hierarchy found in LogScope, logics in JavaMOP are not clearly ordered and complement each other.

5.1.6 *Stream Computation*

All the solutions presented so far check properties on the execution of a system by observing events or state changes of the system. Lola [24] is a runtime monitor for synchronous systems which takes a radically different approach: specifications are written as computations over the stream of values manipulated by a synchronous system. This unique approach allows specifications to resemble synchronous programs. Besides checking logical properties, Lola also supports numeric queries over streams. It was specifically designed for synchronous systems and adapting it for event traces is not obvious. For this reason, we omit Lola in the rest of this chapter.

5.2 COMPARISON AND DISCUSSION

Comparing the expressiveness of various temporal specification formalisms is difficult. For instance, Reger and Rydeheard showed that the relationship between First-Order Linear Temporal Logic for finite traces and parametric trace slicing is nontrivial [65]. Instead of establishing a formal relation between the aforementioned formalisms, we compare them against the key requirements derived by the analysis of TKA (Chapter 3), as well as two additional criteria regarding usability. Table 5.1 summarizes the comparison. In the following, we further detail each characteristic chosen for comparison.

PARAMETRIC EVENTS Supporting parametric events is a critical requirement for the project. Although pattern systems and adaptations of LTL for finite traces do not support parametric events, most temporal specification formalisms designed for runtime verification do.

Table 5.1: Comparison of several temporal specification languages

Language	Parametric	Comp. Values	Quantification	Ref. Past Data	Real-Time	Paradigm
Dwyer's Patterns [28]	✗	n/a	n/a	n/a	✗	declarative
Propel [71]	✗	n/a	n/a	n/a	✗	declarative
RSL [25]	✗	n/a	n/a	n/a	✓	declarative
Salt [9]	✗	n/a	n/a	n/a	✓	declarative
LTL_f [7]	✗	n/a	n/a	n/a	✗	declarative
$TLTL_f$ [8]	✗	n/a	n/a	n/a	✓	declarative
Eagle [2]	✓	✗	global	✓	✗	declarative
Stolz's Param. Prop. [74]	✓	✗	local	✗	✗	declarative
FO-LTL^+ [36]	✓	✓	local	✗	✗	declarative
MFOTL/MONPOLY [6, 16]	✓	✗	global	✓	✓	declarative
JavaMOP [48]	✓	✗	global	✓	✗	mixed
QEA/MarQ [1, 63]	✓	✗	global	✓	✗	operational
Mufin [26]	✓	✗	global	✓	✗	operational
Ruler [4]	✓	✗	n/a	✓	✗	operational
Logfire [38]	✓	✗	n/a	✓	✗	operational
LogScope [3]	✓	✗	global	✗	✗	mixed
ParTraP	✓	✓	local	✓	✓	declarative

COMPOUND VALUES Parametric events may carry compound values such as lists and records. To the best of our knowledge, only a few of the formalisms presented previously support further inspection of compound values. This feature is mandatory in order to exploit traces where event parameters can hold structured data such as records.

LOCAL VS. GLOBAL QUANTIFICATION We can distinguish two types of quantifications in formalisms for parametric monitoring. In *global* quantification the domain value of a quantified variable is defined as the values taken by this variable in a whole trace. On the contrary, in *local*, the quantification domain of a variable may only depend on the current state. Local quantification is mostly useful in combination with support for compound values as it allows quantifying over lists that are carried as event parameters. Only a few formalisms use local quantification, while all approaches based on parametric trace slicing use global quantification.

REFERENCE TO PAST DATA It is well-known that adding past operators to LTL does not increase its expressiveness [35]. However, this result no longer holds when LTL is extended with quantifiers. To see why, consider the following informal specification: "for each value x in the parameter p_1 of an event e_1, there must be an occurrence of an event e_2 before e_1 and with a parameter p_2 equal to x". Because the occurrences of the event e_2 are constrained by the parameters of another event that is yet to occur, this property cannot be captured in a future-only LTL extended to first order. Monitoring such a property is expensive and many specification formalisms do not include past operators for efficiency and simplicity reasons. A possible approach – as taken in QEA [1] – is to explicitly store the data for future use, e.g. the set of values taken by the parameter p_2. Another approach applicable to temporal logics is simply to support past operators, while making sure to handle the complexity increase properly, as demonstrated by MFOTL [6] and its reference implementation MONPOLY [16].

LANGUAGE SUPPORT FOR REAL-TIME Real-time support can be classified according to three levels: unsupported, supported as regular data, supported at language-level. Although not a requirement for the MODMED project where we gathered only a few real-time properties, we argue that supporting time at language-level enables clearer specifications, and more efficient implementation thanks to the monotony of time. For instance, one can stop checking a bounded safety property once the time bound has passed.

PARADIGM Specification formalisms may be classified according to their paradigm: declarative or operational. Even if they both have their strengths, as exemplified by Havelund and Reger [39], we argue

that an operational style introduces additional complexity for the user. For instance, in rule-based system each rule may impact the behavior of the others by adding or removing shared facts, which requires careful attention. This problem is similar to imperative code routines where side-effects interplay is often critical. Moreover, properties are often given in a declarative style (e.g., "*the temperature remains within a certain range*") and a declarative formalism often makes specifications relatable to the properties they encode. It is also possible to mix both styles, as chosen by some formalisms. They offer to encode properties in different styles, usually featuring a declarative and an operational one.

As we can see in Table 3.1, none of the studied formalisms support event parameters with compound values (columns 1 and 2), local quantification over those compound values (columns 3) and referencing past-data (column 4). For this reason, we developed a new language: ParTraP. Its unique combination of characteristics is given at the bottom of the table. The next chapter presents the language in details.

6

THE PARTRAP LANGUAGE

The ParTraP (Parametric Trace Property) language is a new property specification language for finite parametric traces, designed to meet the characteristics of the properties stated in Chapter 3. Although it was mostly influenced by the specification patterns proposed by Dywer et al. [28], it differs from them by being event-based, featuring parametric events, allowing nested scopes and providing timed properties.

This chapter presents ParTraP in details, starting with its trace model in Section 6.1. Then, Section 6.2 describes and illustrates all the features of the language, as well their most useful combinations. Section 6.3 contains many examples of idiomatic ParTraP properties, corresponding to the selected properties from the TKA study. Finally, Section 6.4 classifies the language according to the characteristics used for comparing languages in Chapter 5.

6.1 TRACE MODEL

ParTraP's purpose is to verify trace properties. Therefore, it relies on a trace model that we introduce first. In Section 4.1, we briefly discussed how traces are usually modeled at an abstract level, i.e. as sequences of possibly parametric events. We specialized this abstract view with a particular trace model for ParTraP, which was designed to be simple and flexible enough to support a large variety of data. ParTraP traces are finite sequences of events such that:

1. Events have a *name*. Two events sharing the same name are said to be of the same event *type*.
2. Events have a timestamp, and timestamps must be non-decreasing.
3. Events are parametric, i.e. they may carry data values of interest, called *parameters*.
4. Parameters are key/value pairs, where the key is the parameter name and the value is the associated data.
5. Parameters values can be literals, sequences of values, or records of key/value pairs themselves. Sequences and records allow parameters to be hierarchically structured.

In one sentence, PARTRAP traces are sequences of events that are named, timestamped and parametric, and parameters are records. This model is later formalized in Section 7.3.1.

For illustration purposes, we write traces as a vertical list of events. Each line, or event, takes the form

```
name @ timestamp parameters
```

where parameters are given in JavaScript Object Notation (JSON). For conciseness, timestamps are omitted when irrelevant. For instance, the following shows a simplified extract of a TKA trace:

```
...
RegisterTracker   @ 5  { "type": "F", "id": 0 }
SearchTrackers    @ 6  { "types": ["P", "F"] }
RegisterTracker   @ 7  { "type": "P", "id": 1 }
StartAcquisitions @ 8  {}
MedialMalleolus   @ 9  { "point": [0.5, 1.0, 0.8] },
...
ReplaceTracker    @ 14 { "id": 1 },
RegisterTracker   @ 15 { "id": 2, "type": "P" },
ActivateTracker   @ 16 { "id": 2 },
...
LateralMalleolus  @ 20 { "point": [0.6, 0.9, 0.9] }
...
```

The first event registers a tracker of "`type`" F (attached to the Femur), whose "`id`" is `0`. This event took place at time 5. The second event declares that the current surgery needs trackers P and F. The third event corresponds to the registration of the P tracker (P stands for Pointer). The fourth event records the beginning of the acquisition phase. During this phase, the coordinates of the medial and the lateral malleolus are recorded (at times 9 and 20). Also, from 14 to 16, the pointer is replaced by another pointer which is registered and then activated.

6.2 LANGUAGE FEATURES

6.2.1 *Event Descriptors*

In order to write properties, it is necessary to be able to match specific events. The expressions responsible for this are called *event descriptors*. They can be simple or complex.

SIMPLE DESCRIPTORS At the simplest, event descriptor can be a single name, designating all the events with that particular name. For instance, if `Error` is the name of the event produced upon errors, then the property

```
absence_of Error
```

forbids any `Error` event (the `absence_of` keyword is further detailed later). Additionally, event descriptors allow suffixing the event name with a variable name, such as the variable name `e` in the property

```
absence_of Error e
```

In that case, `e` will be bound to each event named `Error`. While it is completely useless in this particular example, we will see later that it becomes crucial when nesting properties. It also enables another construct: when an event name is suffixed with a variable name, it is possible to add a condition on the event with the `where` keyword, such as in the following property:

```
absence_of Error e where e.cause == "EoF"
```

As each `Error` event will be bound to the variable `e`, we can access the event through `e` in the condition and further constrain it. In this example, the `cause` field must be set to `"EoF"` for an event to match the description, effectively filtering out unwanted errors. Thus, this property only forbids errors whose cause was `"EoF"`. For instance, the following trace satisfies the property as no event matches both the name and the condition of the event descriptor:

```
Exception { "cause": "EoF" }
Error { "cause": "Out of memory" }
```

It is worth mentioning that, if a parameter name is used in an event descriptor, then any event susceptible to match this descriptor must carry this parameter. If this condition is violated, property evaluation will fail. In the above example, this means that `Error` events must have a "`cause`" parameter.

EXPRESSION LANGUAGES Conditions in the `where` clause can be expressed in two different languages. The first one, used in the previous example, is a very simple expression language tailored for ParTraP. It includes common literals, variables, record field accesses with the dot operator, and basic boolean and arithmetic relations. The condition `e.cause == "EoF"` from the previous example property illustrates all of them with

- the string literal `"EoF"`,
- the variable `"e"`,
- the dot operator to access the `"clause"` field of `"e"`, and
- the boolean-valued equality relation.

Although very simple, this language allows expressing the most common constraints on events.

Whenever the simple expression language is too limited, one can resort to Python simply by surrounding the condition with dollar signs.

For instance, we could want to test if the cause starts with `"EoF"`, instead of being strictly equal to it:

```
absence_of Error e where $e.cause.startswith("EoF")$
```

The choice of Python was motivated by its simplicity and its ubiquity in data processing. In particular, some TKA properties rely on three-dimensional geometry, for which there is a mature Python ecosystem.

COMPLEX DESCRIPTORS Event descriptors can also be more complex. It is possible to designate an unordered collection of events with the `set` construct:

```
set(E1 x1, ..., En xn) where c
```

This event set will be triggered after seeing all of the events `E1...En` in any order, provided that they respect the condition `c` when `E1...En` are bound to `x1...xn`, respectively.

6.2.2 *Patterns*

Patterns are the simplest kind of PARTRAP property, and are present in any property. They rule the occurrences of events in a trace. There are two unary patterns:

```
occurrence_of n A
```

where `n` is an optional expression that must evaluate to a positive integer, and its dual:

```
absence_of A
```

The `occurrence_of` pattern requires the occurrence of *at least* `n` events matching the event descriptor `A`. If `n` is omitted, it defaults to `1`. The `absence_of` pattern simply forbids events matching the given event descriptor from occurring. As `A` is an event descriptor, it may consist of a single event name or it can be further constrained with a `where` condition. Examples of pattern satisfaction for simple abstract traces are given in Table 6.1.

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

Table 6.1: Examples of pattern satisfaction for various traces

Following the pattern system proposed by Dwyer et al., ParTraP includes the Response pattern with the infix keyword `followed_by`:

```
A followed_by B
```

This operator takes the same meaning as the original Response pattern: it describes a relationship between a pair of event descriptors where events matching the descriptor `A` must be followed by an event matching the descriptor `B`, and not necessarily immediately. Moreover, events matching `A` are not required to occur. ParTraP also includes its converse, the Precedence pattern, with the infix keyword `preceded_by`:

```
A preceded_by B
```

It also embodies the same definition as the original Precedence pattern: occurrences of events matching the descriptor `B` are a necessary pre-condition for the occurrence of an event matching the descriptor `A`. Events matching `B` are not required to occur. Table 6.1 illustrates the difference between the Response and Precedence patterns. Besides those two binary patterns proposed by Dwyer et al., ParTraP includes a third binary pattern:

```
A prevents B
```

As the `prevents` keywords indicates, this property forbids the occurrence of events matching the descriptor `B` after the occurrence of an event matching the descriptor `A`.

These patterns differentiate themselves from the original patterns of Dwyer et al. with their support for parametric events. Indeed, the event descriptor on the right-hand side may depend on the event matched on the left-hand side, enabling the definition of relationships between events according to their parameters. Let us illustrate this particular feature with an example. Consider the property that any user logging in a system must eventually log out. These actions are observed with the events `Login` and `Logout`, which both contain a `uid` parameter corresponding to the user's unique identifier. This property can be captured elegantly with the Response pattern combined ParTraP's support for event parameters:

```
Login in followed_by Logout out where out.uid == in.uid
```

For instance, the following trace containing the login and logout events of several users satisfies the property:

```
Login  { "uid": 0 }
Login  { "uid": 1 }
Login  { "uid": 0 }
Logout { "uid": 0 }
Logout { "uid": 1 }
Logout { "uid": 2 }
```

Observe that user 2 is able to logout even though they never logged-in, or that user `0` is able to login twice. The response pattern only enforces disconnection of connected users and does not prevent those two behaviors.

6.2.3 *Scopes*

The range of a trace where a ParTraP property should hold can be restricted through *scopes*, which are delimited by events. In the absence of scope restrictions, properties must hold on the whole trace. Scoped properties are also properties, which enables nesting several scope restrictions. Scopes can be classified according to their arity, i.e. the number of events they involve. The 6 unary scopes of ParTraP, illustrated in Figure 6.1, are versatile combinators.

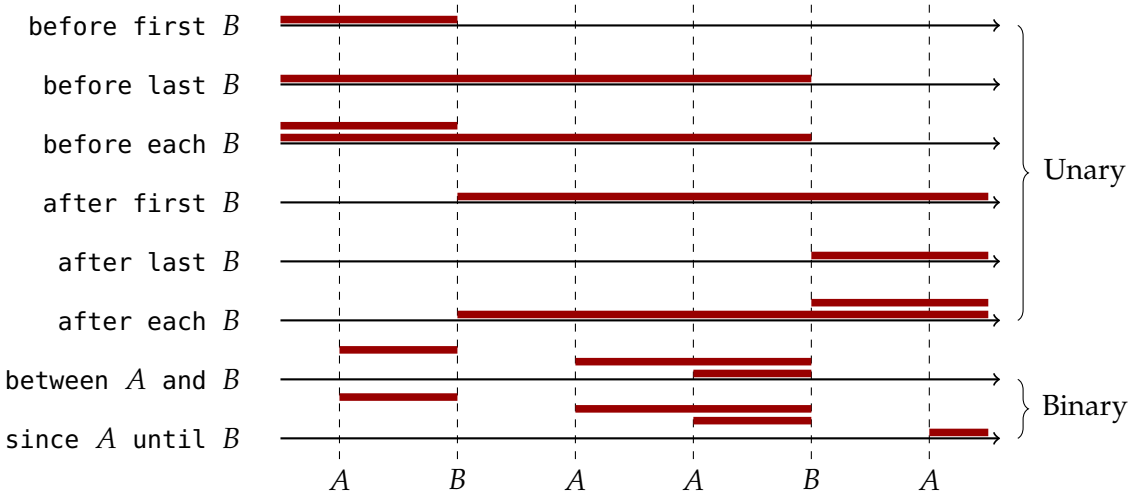


Figure 6.1: Graphical representation of ParTraP scopes

Unary scopes restrict the range of a trace where a property should hold to what comes after an event matching a given event descriptor (`after`), or before an event matching a given event descriptor (`before`). As several events may match an event descriptor, both directions must be further refined with an occurrence specifier: `first`, `last` or `each`. This effectively creates the 6 possibles combinations depicted in Figure 6.1. For instance, the ParTraP property

```
before first Login, absence_of NewMessage
```

captures the property that a system should not receive any message until somebody is logged-in. The comma delimits the scope from the inner property. More generally, commas are consistently used to wrap a property under a specific context, forming a new property. The following trace satisfies the above property example:

```
Login      { "uid": 0 }
Login      { "uid": 0 }
NewMessage {}
```

```
Login      { "uid": 0 }
```

Similarly to binary patterns where both events may be related when associating them to a variable name, if an event descriptor for scopes is suffixed with a variable name, the delimiting event will be made available under that name in the underlying property. This mechanism makes the `each` variants of unary scopes especially versatile. These variants ensure that a property is true after, or before, each and every event occurrence matching the given event descriptor. For instance, consider the previous example saying that a system should not receive a message before a user is logged-in. If the system allows multiple users at the same time, we must distinguish from who the message was from, and whether this user is logged-in. The new property can be expressed informally as *"the system should not receive any message from a user until that same user is connected"*. Assuming that the event `NewMessage` now also carries the user's unique identifier, the refined property can be specified with the following ParTraP expression:

```
before each Login in, absence_of NewMessage msg where msg.uid ==
    in.uid
```

The following trace violates this new property whereas it satisfies the old one:

```
Login      { "uid": 0 }
NewMessage { "uid": 0 }
Login      { "uid": 1 }
NewMessage { "uid": 0 }
NewMessage { "uid": 2 }
Login      { "uid": 2 }
```

Indeed, while users 0 and 1 match the expected behavior, the system received a message from user 2 before they logged-in, which the property forbids. Note that although the `each` variant may lead to evaluating the same underlying property on overlapping intervals, such as pictured in Figure 6.1, the outcome may be different for each interval as they can depend on the delimiting events. This is the case in the previous example, where the invalid scope for user 2 is overlapping with the valid scopes for user `0` and 1. By delimiting the range of a trace where a property should hold, and providing access to the parameters of a delimiter event at the same time, scopes allow to concisely express temporal relations directed by event parameters.

The `each` variants also happen to be useful combinators. For instance, we can express that the property `P` should be true for all the segments of a trace between an event `A` and an event `B` by nesting two scope expressions:

```
after each A, before first B, P
```

For convenience, this scope is included in PARTRAP as `between A and B`. Similarly to the original specification patterns of Dwyer et al., we also included its weak variant: `since A until B`. This variant does not require for the right-hand side event (`B`) to occur, in which case the last interval is only half-bounded. This difference is illustrated in Figure 6.1.

As a consequence of being defined on top of nested scopes, the event delimiting the end of a binary scope may depend on the one starting it. To illustrate binary scopes, consider another example: "*a user should not log-in twice*". We can use a binary scope to ensure that, since the moment a user logs in, and until the moment they log out, they should not be able to log-in a second time:

```
since Login in1 until Logout out where out.uid == in1.uid,
    absence_of Login in2 where in2.uid == in1.uid
```

It exploits the fact that the ending event (`Logout`) may depend on the starting one (`Login`) to pair connections and disconnections from a same user. The following trace example contains the events regarding two users:

```
Login      { "uid": 1 }
Login      { "uid": 0 }
NewMessage { "uid": 0 }
Logout     { "uid": 0 }
Login      { "uid": 1 }
NewMessage { "uid": 1 }
Login      { "uid": 0 }
Logout     { "uid": 0 }
```

User `0` logs in several times, but only after disconnecting, which matches the expected behavior. On the contrary, user `1` was able to log-in twice and causes the property to be violated. This trace is a good example for also illustrating the difference between the two binary scopes. Had we used the `between/and` scope, the violation would not have been detected. Indeed, the latter only enforces its underlying property on scopes that delimited by both a starting and an ending events. In this case, user `1` never logs out and no `between/and` scope would have existed.

Finally, two design decisions are worth mentioning:

- All scopes are open on both sides, i.e., delimiter events are not included in the interval they define. The symmetry makes the `after` and `before` scopes consistent.
- If the event delimiting a scope never occurs, the scope does not exist and the whole property is vacuously true. This view is similar to universally quantified formulas in first-order logic, which are always true when the quantification domain is empty.

6.2.4 *Timed Variants*

Unary scopes and binary patterns may be additionally constrained by a duration expressed in common time units (`h`, `s`, etc.). This mechanism relies on the fact that events are timestamped.

Unary scopes can be prefixed with the `within` keyword and a duration expression, such as in the following abstract property:

```
within 2ms before each A, absence_of B
```

The inner property only has to hold for the given duration starting immediately at the delimiter event for the `after` scope, or ending exactly at the delimiter event in the `before` case. In the previous example, the event *B* should not occur during the two milliseconds preceding any occurrence of an event *A*. The following trace violates the property because the second event `A` occurs within two milliseconds of the second event `B`:

```
A @ 2.018 {}
B @ 2.025 {}
A @ 2.028 {}
B @ 2.029 {}
B @ 2.033 {}
```

Binary patterns may also be suffixed with `within` and a duration expression. For instance, the response pattern becomes bounded in time:

```
A followed_by B within 2s
```

6.2.5 *Quantifiers*

The trace format allows compound values in event parameters. In particular, they can be lists of values. ParTraP allows exploiting them with quantified properties.

The universal quantifier takes the following form:

```
forall x in L, P
```

where `x` is an identifier and `L` is a list. It is defined as in first-order logic, that is the property `P` must be satisfied for each possible value of `x` taken from the list `L`. The existential quantifier `exists` is also defined as usual.

Since quantified properties are themselves properties, they can be arbitrarily nested. In particular, it is convenient to use a quantifier inside a scope property: if a parameter of the event delimiting the scope is a list, it can be used as the quantification domain. For instance, assume that we have a system that can be asked to list all the users who ever logged-in. Then we can verify that those users actually logged-in in the past:

```
before each UserList users,
    forall uid in users.uids,
        occurrence_of Login in where in.uid == uid
```

The following trace example containing two user lists satisfies the property:

```
Login { "uid": 0 }
Login { "uid": 1 }
Login { "uid": 3 }
UserList { "uids": [3, 0] }
Login { "uid": 2 }
UserList { "uids": [2, 0, 3] }
```

Note that this property does not verify that the user list is exhaustive, which would be a different property. In particular, user `1` was not accounted for.

6.2.6 *Event Selection*

The only way to extract the parameters of an event so far is to bind the event in a scope. However, the associated restriction of the trace range might not be desired. PARTRAP has the dedicated "`given`" expression for this purpose. It takes the same syntactic form as scopes, i.e. suffixed with an occurrence specifier (`first`, `last` or `each`) and an event descriptor. It wraps another property that will be evaluated in an environment extended with the selected event. In the `each` case, the property must be true for all events matching the event descriptor. Like scopes, a property constructed with `given` will be true if no event matches the descriptor.

This construct was actually added to the language because of real need in the TKA case study. The property that conducted its inception best demonstrates its usage. It can be stated as "*All the necessary trackers are registered before entering the state TrackersVisibCheck*", where the necessary trackers are given by the `SearchTrackers` event. The particularity was that trackers can be registered either before or after that the search for them begins. The "`given`" expression is used to extract the list of necessary trackers, without constraining their registration to the past or the future:

```
before each EnterState e where e.state == "TrackersVisibCheck",
    given last SearchTrackers st,
        forall ty in st.types,
            occurrence_of RegisterTracker rt where rt.type == ty
```

The following shows a simplified extract of a TKA trace satisfying this property:

```
RegisterTracker { "type": "F", "id": 0 }
SearchTrackers  { "types": ["P", "F"] }
RegisterTracker { "type": "P", "id": 1 }
SearchTrackers  { "types": ["P", "F", "T"] }
```

```
RegisterTracker { "type": "T", "id": 2 }
EnterState      { "state": "TrackersVisibCheck" }
```

6.2.7 *Logical Operators*

Properties can be logically connected with the usual logical operators: `not`, `or`, `and`, `implies`, and `equiv`. Note that events bound to variables in one side of one of those binary operators is not be available in the other side. For instance, in the property

```
(after first A x, P) or (absence_of B b where c)
```

the variable `x` will not be accessible in the condition `c`. This allows preserving the usual properties of logical operators, such as commutativity of the disjunction. Moreover, the negation operator `not` has the same precedence as scopes and quantifiers, while binary logical operators have a lower precedence than any other construct in ParTraP. Thus, the parentheses in the previous example are not required, but kept for clarity.

6.3 SPECIFICATION EXAMPLES: TKA PROPERTIES

The following list of examples illustrates the idiomatic expression of the selected TKA properties (Chapter 3) with ParTraP. Each property is restated for convenience, and then followed by its ParTraP expression.

Property 1. *The trace contains a step "redo acquisitions".*

Operational steps are encoded as state transitions in TKA software, and redoing the acquisitions has a corresponding state that can be looked for:

```
occurrence_of EnterState e where e.state == "RedoAcquisitions"
```

Property 2. *The temperature of the camera stays within a given interval.*

Assuming that the interval is $[l, u[$, we can make sure the temperature never goes out of those bounds:

```
absence_of Temp t where not (l <= t.t1 and t.t1 < u)
```

Property 3. *The distance between pairs of hip centers is less than d.*

Each successful acquisition of a hip center produces a `HipCenter` event carrying the acquired position in its `point` parameter. Even though this property is not temporal, we can encode it using a scope:

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

where `dist` is an external Python function returning the euclidean distance between two given points.

Property 4. *The distance between the hip center and the knee center is greater than d.*

```
absence_of set(HipCenter hc, KneeCenter kc)
    where $dist(hc.point, kc.point) <= d$
```

where `dist` is the same function as in the previous property.

Property 5. *If the medial malleolus is farther from the camera than the lateral one, a warning is issued.*

As hip and knee center events, malleoli events carry the acquired position in their `point` parameter.

```
set(MedialMalleolus m, LateralMalleolus l)
    where $norm(l.point) < norm(m.point)$
followed_by WarningMalleolusInverted
```

where `norm` is an external Python function returning the norm of a vector.

Property 6. *The user never skips a screen.*

We arbitrarily define 200 milliseconds as the threshold below which a screen is considered skip. The event `ActionNext` is triggered whenever the user asks to move to the next step and must not occur for 200 milliseconds after entering a new state:

```
EnterState prevents ActionNext within 200 ms
```

Property 7. *The acquisition of a point succeeds if and only if the probe is stable.*

Whenever a new point cloud is acquired, the system produces an event `NewPointCloud` carrying the set of points. This event must be followed by a `PointAcquired` event before the next point cloud if it was stable, and prevents point acquisition if it was not stable.

```
since NewPointCloud pc where $isStable(pc.cloud)$
until NewPointCloud,
    occurrence_of PointAcquired
and
since NewPointCloud pc where $not isStable(pc.cloud)$
until NewPointCloud,
    absence_of PointAcquired
```

where `isStable` is an external Python predicate that holds if the given set of points is cohesive enough.

Property 8. *The protocol "redo acquisitions" only proposes already performed acquisitions.*

When the step "redo acquisitions" is reached, the system produces an event `RedoOptions` carrying the list of options offered to the user. These options must correspond to steps that were visited in the past:

```
before each RedoOptions r,
    forall option in r.options,
        occurrence_of EnterState e where e.state == option
```

Property 9. *Detecting a new tracker produces a dialog asking for replacement confirmation.*

```
after each RegisterTracker rt,
    TrackerDetected td where td.type == rt.type
    followed_by DialogConfirmReplace dc where dc.type == rt.type
```

Property 10. *The state TrackersConnection is unreachable until the camera is connected.*

```
EnterState e where e.state == "TrackersConnection"
preceded_by CameraConnected
```

Property 11. *A replaced tracker is not used until it is registered again.*

```
since Unregister unr until Register reg where reg.id == unr.id,
    absence_of Activate act where act.id == unr.id
```

Property 12. *The action "previous" cancels the current point cloud acquisition.*

A cancelled acquisition must never be followed by a success, at least until a new acquisition is started:

```
since ActionPrevious until AcquisitionBegin,
    absence_of AcquisitionSuccess
```

Property 13. *All the necessary trackers are seen before entering the state TrackersVisibCheck.*

This property and its encoding were discussed in Section 6.2.6.

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

Property 14. *On the tracker connection screen, a tracker is shown if and only if it is necessary.*

A possible way to formalize this property is to ensure that the list of trackers shown is always the one that was declared in the previous event `SearchTrackers`:

```
since SearchTrackers st1 until SearchTrackers,
    absence_of TrackersConnectionList stc
        where stc.types != st1.types
```

Property 15. *In the state TrackersConnection, not detecting a single tracker for 2 minutes produces an error message.*

Unfortunately, ParTraP's timed operators are not useful in the formalization of this particular physical time requirement. Nonetheless, one can resort to comparing timestamps as data parameters:

```
since EnterState enter where enter.state == "TrackersConnection"
until ExitState exit where exit.state == enter.state, (
    (absence_of TrackerDetected implies ErrorNoTrackerDetected)
    and
    TrackerDetected td where td.time > enter.time + 120
    preceded_by ErrorNoTrackerDetected
)
```

The `since/until` scope captures the fact that the property only applies to the state TrackersConnection. In this scope, the property covers two scenarios: either no trackers are detected, in which case an error message must occur, or a tracker is detected later than 2 minutes after entering the state TrackersConnection, in which case an error message must have occurred by the past.

6.4 CHARACTERISTICS

In Chapter 5, we compared several specification formalisms on 6 criteria, and showed where ParTraP fits in this comparison at the bottom of Table 5.1. We now come back to ParTraP's classification for each of the 6 criteria:

PARAMETRIC Parameters values of an event can be inspected and used in computations or relations thanks to "`where`" conditions, which are used everywhere an event is described in a property. In particular, those conditions allow relating different events according to their parameters.

COMPOUND VALUES ParTraP's trace model does not restrict to the nature of parameters to atomic values. The dot operator in simple expressions allows accessing fields of records, recursively if necessary. The extracted value can be used in simple relations, or as quantification domain if it is a list. Besides simple expressions, one can also use Python to access structured parameters.

QUANTIFICATION The quantification domain of the `forall` and `exists` operators only depends on the current environment where they are evaluated. This choice allows using parameters values as quantification domains. Most often, the quantification domain is a parameter from an event matched earlier in the property. Quantification in ParTraP is limited to the current state of the evaluation, and therefore said to be local.

REFERENCE TO PAST DATA Because ParTraP operates on whole trace intervals there is no notion of "current" instant. In consequence, there is no notion of past or future. Unless there

are explicitly sliced out with a scope, events and their data are always accessible from anywhere.

REAL-TIME We have seen that PARTRAP features the operator `within`, allowing to express real-time durations with common time units (e.g., h, s). It limits the duration of a scope or add time constraints to binary operators.

PARADIGM PARTRAP is completely declarative. Properties describe *what* should hold, and not *how* to ensure that it does. In particular, the user does not manage a state, which is typical of the operational style. Although properties are evaluated in an environment, which could be assimilated to a state, this state is always updated implicitly according to the property formulation.

This unique combination of features makes PARTRAP a novel approach on trace property specification, especially suited to traces containing data-rich and structured events.

The next chapter defines PARTRAP's syntax and semantics formally.

7

FORMAL DEFINITION OF PARTRAP

In this chapter, we formalize both the syntax and semantics of ParTraP. This is especially important as certain expressions resemble English sentences, which would leave room for personal interpretation if not properly formalized.

7.1 EXPRESSION LANGUAGE

Some elements of the language such as the `where` clause rely on predicate expressions. Since those expressions are orthogonal to the core definition of ParTraP, we will first introduce them in this dedicated section.

It is useful to be able to define simple to moderately complex expressions in properties, while delegating more complex expressions to an external language. As already mentioned, we chose Python for this role.

Let Expr be the set of expressions that can be derived from the rule ⟨*expr*⟩ from the grammar in Figure 7.1, written in Extended Backus–Naur Form (EBNF) [68]. To formalize expression evaluation,

⟨*expr*⟩	::=	⟨*literal*⟩	
	|	⟨*unop*⟩ ⟨*expr*⟩	
	|	⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩	
	|	'`(`' ⟨*expr*⟩ '`)`'	
	|	⟨*ident*⟩	(variable lookup)
	|	⟨*expr*⟩ '`.`' ⟨*ident*⟩	(record field access)
	|	⟨*expr*⟩ '`[`' ⟨*expr*⟩ '`]`'	(sequence indexing)
	|	'`$`' (any character but '`$`')* '`$`'	(Python expression)
⟨*unop*⟩	::=	'`not`' | '`-`'	
⟨*binop*⟩	::=	'`<`' | '`<=`' | '`==`' | '`>`' | '`>=`' | '`!=`' | '`&&`' | '`||`' | '`+`' | '`-`' | '`*`' | '`/`' | '`%`'	
⟨*ident*⟩	::=	set of alphanumeric identifiers	
⟨*literal*⟩	::=	usual set of booleans, integers, floating-points and string literals	

Figure 7.1: Syntax of expressions

we will use the judgement form $\eta \vdash e \downarrow v$, read as "in the environment η, expression e reduces to value v". Evaluation of basic expressions (literals, unary expressions and binary expressions) is defined as usual and not detailed here. The interesting rules are the following:

$$\frac{\eta(x) = v}{\eta \vdash x \downarrow v}\ \text{E-Lookup}$$

$$\frac{\eta \vdash e \downarrow r \quad r(p) = v}{\eta \vdash e.p \downarrow v}\ \text{E-FieldAccess}$$

$$\frac{\eta \vdash e_1 \downarrow s \quad \eta \vdash e_2 \downarrow i \quad s_i = v}{\eta \vdash e_1[e_2] \downarrow v}\ \text{E-Indexing}$$

$$\frac{}{\eta \vdash \texttt{\$ PythonExpr \$} \downarrow v}\ \text{E-PythonCall}$$

The rule E-Lookup resolves variables names to their values in the current environment, E-FieldAccess accesses the value associated a key in a record, and E-Indexing addresses the element at a given index in a sequence. The last rule, E-PythonCall, invokes Python on an expression and treats it as a black box returning any value.

7.2 SYNTAX

The syntax of ParTraP is defined by the grammar in Figure 7.2. This is an abstract grammar as it does not capture the precedence between the operators manipulating properties, nor their associativity. For instance, the string "`not` P_1 `and` P_2" could be parsed as "`not (`P_1 `and` P_2`)`" or "`(not` P_1`) and` P_2". Any grammar allowing the derivation of more than one parse tree for a same input string is called an *ambiguous grammar*. While ambiguous, this abstract view is useful to understand the syntax of ParTraP as it contains all the language constructions.

Table 7.1: Precedence of ParTraP operators on properties

Precedence	Operators
5	`not`, `given`, quantifiers and scopes
4	`and`
3	`or`
2	`implies`
1	`equiv`

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

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

⟨*occ*⟩ ::= ‘each’ | ‘first’ | ‘last’

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

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

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

Figure 7.2: Syntax of ParTraP

Instead of showing the non-ambiguous, yet inscrutable concrete syntax, we disambiguate the abstract grammar by specifying operators precedence and associativity. The precedence level of the property operators are given in Table 7.1. A higher precedence number corresponds to a higher precedence level. The logical operators follow a commonly used precedence [59]. The `given` construct, the scopes (both unary or binary) and the quantifiers can be seen as unary prefix operators on properties. Together with the `not` operator, they are all given the same and highest precedence level. Thus, unary operators only apply to the property directly following it. Following those rules, the example string “`not` P_1 `and` P_2” is equivalent to “`(not` P_1`)` `and` P_2” in ParTraP. Properties can be grouped explicitly under the same unary operator with parentheses. Finally, all binary operators are left associative, even the implication, which is usually right associative. Precedence and associativity rules keeps the language simple and coherent.

7.3 SEMANTICS

This section formalizes the semantics of ParTraP. It is defined in terms of inference rules over traces instead of translating it into an existing formalism because there is no well-established formalism for parametric specifications.

For pedagogical reasons, we describe the semantics of all ParTraP constructs except event sets. This construct adds a layer of complexity but does not change the global shape of the semantic rules. The simpler and shorter version presented here is close to the complete rules, which can be found in the language reference [12]. The additional complexity mainly comes from the two following points:

1. Contrary to single events, event sets last in time; scopes must be updated accordingly.
2. The `first` and `last` event sets matching a set description must be defined carefully.

7.3.1 *Preliminary Definitions*

We use $X \to Y$ and $X \rightharpoonup Y$ to denote the set of total and partial functions from X to Y, respectively. We write finite maps (partial functions over a finite domain) as $[x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ and the empty map as $[\,]$. We note $m[y \mapsto v]$ the map which is the same as m except that the mapping for y is updated to refer to v:

$$m[y \mapsto v](x) = \begin{cases} v & \text{if } x = y \\ m(x) & \text{otherwise.} \end{cases}$$

If S is a set, S^* is the set of *finite* sequences of elements of S.

Equipped with these notations, we may now formalize traces content and format. The set of *values* is the smallest set Val such that:

1. literals (booleans, integers, strings and floating-point numbers) are values;
2. if $v_1, \ldots, v_n$ are values, then the sequence $(v_k)_{k=1}^n$ is a value;
3. if $v_1, \ldots, v_n$ are values and $f_1, \ldots, f_n$ are names, then the map, or record, $[f_1 \mapsto v_1, \ldots, f_i \mapsto v_i]$ is a value.

An *environment* is a map from variable names to values:

$$\text{Env} = \text{Var} \rightharpoonup \text{Val},$$

where Var is the set of variable names.

An *event* is characterized by a *name*, an occurrence time and a set of named *parameters*. Formally an event is defined as a triplet:

$$\text{Event} = \Sigma \times \mathbb{R} \times (P \rightharpoonup \text{Val}),$$

where Σ and P are finite sets of event names and parameter names, respectively. Note that this definition permits events to have the same name and yet different parameters. This provides more flexibility with the input traces and allows, for instance, to have optional parameters. For convenience, we define the three following projections on an event $e = \langle \sigma, t, p \rangle$: $\text{name}(e) = \sigma$, $\text{time}(e) = t$ and $\text{param}(e) = p$.

Finally, a *trace* is a sequence of events $(e_i)_{i=1}^n$ with non-decreasing occurrence times. We can formally define the set of possible traces as follows:

$$\text{Trace} = \{\, (e_i)_{i=1}^n \in \text{Event}^* \mid \forall i \in [1\,..\,n-1],\ \text{time}(e_i) \leq \text{time}(e_{i+1}) \,\}.$$

In the following, traces are also denoted as τ when the indices are irrelevant.

7.3.2 *Events Extraction and Time Slicing*

Finding events that satisfy some constraints expressed in ParTraP properties is a basic necessity to define the semantics of the language. We first introduce a dedicated function that handles that matter. The semantic rules of ParTraP are built upon that function and focus on the temporal aspect.

The function M computes the events of a trace that *match* an event description with a name and a condition on the event, and respect an occurrence specifier, i.e. an element of $\{\texttt{first}, \texttt{last}, \texttt{each}\}$. More precisely, given a trace $(e_i)_{i=1}^m$, $M((e_i)_{i=1}^m, \sigma, x, c, \eta, o)$ is the set of indices i of the trace such that:

- e_i has the name σ,
- the condition c evaluates to **true** in the environment η extended with x associated to e_i, and
- e_i respects the occurrence specifier o.

This set can be computed in two steps: finding M_{desc}, the set of all the indices that match the event description, and then selecting the ones that respect the occurrence specifier. The definition of the function M is based on that idea:

$$M((e_i)_{i=1}^n, \sigma, x, c, \eta, o) = \begin{cases} M_{\text{desc}} & \text{if } o = \texttt{each} \\ \{\min\ M_{\text{desc}}\} & \text{if } o = \texttt{first} \text{ and } M_{\text{desc}} \neq \emptyset \\ \{\max\ M_{\text{desc}}\} & \text{if } o = \texttt{last} \text{ and } M_{\text{desc}} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where

$$M_{\text{desc}} = \{j \in [1\,..\,n] \mid \text{name}(e_j) = \sigma \wedge \eta[x \mapsto \text{param}(e_j)] \vdash c \downarrow \textbf{true}\}.$$

M_{desc} is the result of the first step, i.e. it is the set of indices that match the event description given by the name σ and the condition c. The subset of M_{desc} that is actually returned is computed according to the occurrence specifier o. For instance, if $o = \texttt{first}$, only the minimal index in M_{desc} is returned, which indeed corresponds to the first event of the trace that matches the description.

We also need the ability to slice a trace according to a time limit. If $(e_i)_{i=1}^m$ is a trace and l is a real, the function

$$\text{upto}((e_i)_{i=1}^m, l) = (e_i)_{i=1}^{\max(\{\, j \in [1..m] \mid \text{time}(e_j) < l \,\} \cup \{0\})}$$

slices the trace $(e_i)_{i=1}^m$ from its beginning and up to the time limit l. The union with the singleton $\{0\}$ in the upper bound ensures that an empty sequence is returned if there are no events occurring before the time limit. The symmetrical operation, i.e. slicing a trace from a time point up to its ending, is performed by the following function:

$$\text{since}((e_i)_{i=1}^m, l) = (e_i)_{i=\min(\{\, j \in [1..m] \mid \text{time}(e_j) \geq l \,\} \cup \{0\})}^m.$$

7.3.3 *Semantic Rules*

The semantics of the original pattern system proposed by Dwyer et al. was given through translation rules manually defined for each pair of pattern and scope. Because of the number of combinations, there are numerous rules and they have been shown to be inconsistent by Taha et al. [75]. Since ParTraP scopes can be arbitrarily nested, the number of combinations is infinite and defining an exhaustive set of rules is impossible. Instead, property satisfaction is defined through recursive rules derived from their informal meaning, and where each rule only handles a single construct.

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 9 inference rules. The satisfaction relation for disjunction and negation is given by the rules Disj and Neg, respectively:

$$\frac{\tau \vDash_\eta P_1 \vee \tau \vDash_\eta P_2}{\tau \vDash_\eta P_1 \texttt{ or } P_2}\ \textsc{Disj} \qquad \frac{\neg(\tau \vDash_\eta P)}{\tau \vDash_\eta \texttt{not } P}\ \textsc{Neg}$$

They are straightforward and require no further explanation. Universal quantification is covered by the Forall rule:

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

This rule 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. Satisfaction of event occurrence is given by the rule OCC:

$$\frac{\eta \vdash n_e \downarrow n \quad |M(\tau, \sigma, x, c, \eta, \texttt{each})| \geq n}{\tau \vDash_\eta \texttt{occurrence\_of}\ n_e\ \sigma\ x\ \texttt{where}\ c}\ \textsc{Occ}$$

It asserts the occurrence of a particular event description by measuring the size of the set returned by the M function, i.e. counting the number of events that match this description in the current trace, and checking that it is greater or equal to the computed value of n_e. The rule for the `after` is more interesting:

$$\frac{\forall j \in M(\tau, \sigma, x, c, \eta, o),\ (\tau_i)_{i>j} \vDash_{\eta[x \mapsto \text{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{after}\ o\ \sigma\ x\ \texttt{where}\ c,\ P}\ \textsc{Aft}$$

It relies on the results of the M function to slice the trace after the end of each event matching the description and to update the evaluation environment for the underlying property. The timed variant of the `after` scope is covered by the AFTT rule, which generalizes the AFT rule with an additional time bound:

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall j \in M(\tau, \sigma, x, c, \eta, o),\ \text{upto}((\tau_i)_{i>j}, \text{time}(\tau_j) + \delta) \vDash_{\eta[x \mapsto \text{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{within}\ \delta_e\ \texttt{after}\ o\ \sigma\ x\ \texttt{where}\ c,\ P}\ \textsc{AftT}$$

It is similar to its predecessor with the addition that it evaluates a duration expression and slices the trace so that it lasts at most for this duration. Rules for the `before` scope and its timed variant are symmetrical to the ones for the `after` scope:

$$\frac{\forall j \in M(\tau, \sigma, x, c, \eta, o),\ (\tau_i)_{i<j} \vDash_{\eta[x \mapsto \text{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{before}\ o\ \sigma\ x\ \texttt{where}\ c,\ P}\ \textsc{Bef}$$

$$\frac{\eta \vdash \delta_e \downarrow \delta \quad \forall j \in M(\tau, \sigma, x, c, \eta, o),\ \text{since}((\tau_i)_{i<j}, \text{time}(\tau_j) - \delta) \vDash_{\eta[x \mapsto \text{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{within}\ \delta_e\ \texttt{before}\ o\ \sigma\ x\ \texttt{where}\ c,\ P}\ \textsc{BefT}$$

The last rule handles the `given` expression:

$$\frac{\forall j \in M(\tau, \sigma, x, c, \eta, o),\ \tau \vDash_{\eta[x \mapsto \text{param}(\tau_j)]} P}{\tau \vDash_\eta \texttt{given}\ o\ \sigma\ x\ \texttt{where}\ c,\ P}\ \textsc{Given}$$

It is a simplified variant of the AFT rule that does not slice the trace. Finally, we say that a trace τ *satisfies* a property P when $\tau \vDash_{[]} P$.

7.3.4 *Derived Constructs*

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

- P_1 `and` P_2 = `not (not` P_1 `or not` P_2`)`
- P_1 `implies` P_2 = `not` P_1 `or` P_2
- P_1 `equiv` P_2 = (P_1 `implies` P_2) `and` (P_2 `implies` P_1)
- `exists` x `in` e`,` P = `not forall` x `in` e`, not` P

and the additional temporal expressions:

- `absence_of` E = `not occurrence_of` E
- A `followed_by` B `within` δ =
  `within` δ `after each` A`, occurrence_of` B
- A `preceded_by` B `within` δ =
  `within` δ `before each` A`, occurrence_of` B
- A `prevents` B `within` δ =
  `within` δ `after each` A`, absence_of` B
- `between` A `and` B`,` P = `after each` A`, before first` B`,` P
- `since` A `until` B`,` P = `(between` A `and` B`,` P`) and (`
  `(occurrence_of` B `and after last` B`, after each` A`,` P`)`
  `or (absence_of` B `and after each` A`,` P`)`
  `)`

With the exception of the last one, those definitions are straightforward. The last one, `since` A `until` B`,` P is more complicated as it requires P to be true in-between A and B, but also after each A occurring after the very last B. Each of those A marks the beginning of a right-opened interval where P should hold, matching the idea that B does not have to occur for the scope to exist. If B does not occur at all in the trace, P must simply be true after each A.

Finally, in order for properties to match the semantic rules that use a fully qualified form, optional values are filled as follows:

- events that are not explicitly associated to a variable name are bound to the empty variable name, and
- omitted `where` conditions on events default to `true`.

For instance, in the expression

```
occurrence_of A
```

the event `A` is not associated to a variable name, nor to a "`where`" condition. The Occ rule requires both, and thus the previous expression conceptually defaults to

```
occurrence_of A "" where true
```

Note that the empty variable name `""` is invalid in ParTraP's syntax, and is used here for illustration.

8

PARTRAP IMPLEMENTATION AND EXPERIMENTS

An important goal of the MODMED project was to deliver tools to accompany the proposed specification language. This was crucial for the language to be adopted. These tools rely on a simple trace format (Section 8.1) that is a concrete encoding for the abstract trace model used to define ParTraP in the previous chapters. There are two publicly available implementations of ParTraP using this trace format: an interpreter that I wrote to experiment with the language and its possible implementations (Section 8.2), and an IDE (Section 8.3). I contributed to the design and development of the latter, but I was not the main developer.

8.1 TRACE FORMAT

The trace format was designed to be as simple as possible, so that the tools developed for the MODMED project can also be easy to use in other contexts. Because of simplicity and ubiquity, we chose to encode traces as JSON files [23] that contain a single array of events. Each entry in the array corresponds to an event occurrence and must be a JSON record, with only a single mandatory key `"name"` to which is associated the event name as a string. If one or several timed operators are used in a property, then events must also be timestamped with a JSON number associated to the `"time"` key. Obviously, timestamps must be non-decreasing. Besides the two aforementioned reserved keys, any other couple key/value in the record is be treated as an event parameter, named according to the key and whose value can be any JSON object. Figure 8.1 shows the JSON encoding of the example trace introduced in Section 6.1.

Blue Ortho traces use an ad-hoc format with little to no structure. In consequence, exploiting events and their data is inconvenient. Thus, we wrote a simple formatter responsible for extracting events of interest and their parameters, and producing a traces complying with our format. In the future, Blue Ortho plans to use structured tracing library developed by MinMaxMedical for the project, which was mentioned in Chapter 2. It support several representation formats, one of them being JSON. It is similar and compatible with the format described above.

```
[
    ...
    { "time": 5,  "name": "RegisterTracker",  "type": "F", "id": 0 },
    { "time": 6,  "name": "SearchTrackers",   "types": ["P", "F"] },
    { "time": 7,  "name": "RegisterTracker",  "type": "P", "id": 1 },
    { "time": 8,  "name": "StartAcquisitions" },
    { "time": 9,  "name": "MedialMalleolus",  "point": [0.5, 1.0, 0.8] },
    ...
    { "time": 14, "name": "ReplaceTracker",   "id": 1 },
    { "time": 15, "name": "RegisterTracker",  "id": 2, "type": "P" },
    { "time": 16, "name": "ActivateTracker",  "id": 2 },
    ...
    { "time": 20, "name": "LateralMalleolus", "point": [0.6, 0.9, 0.9] }
    ...
]
```

Figure 8.1: Example of JSON trace

8.2 COMMAND-LINE INTERPRETER

As soon as a first proposal for ParTraP was drafted, I implemented a Command-Line Interpreter for it. Its interface is a simple read–eval–print loop which parses a property, evaluates it on the traces given as a program arguments and indicates which traces satisfy the property and which do not. In the latter case, a simple yet comprehensive explanation message is also printed. The interpreter is implemented in Haskell because of its efficiency for implementing languages, mostly thanks to its parser combinator libraries [41]. Most of the language is supported, with the exception of Python expressions. There are several libraries for evaluating Python expression from Haskell, but they all seem unmaintained and out of date. Because this implementation was not meant to be used by end-users anyway, we preferred to skip this feature that is mainly targeted for real-world usage, and focus on ParTraP itself.

As all the other tools developed by MODMED partners, the Command-Line Interpreter source code is available online[1] under the GNU Lesser General Public License (LGPL). The repository contains the program sources, a comprehensive test suite and usage instructions.

Having an early implementation helped in many stages of the language development:

1. First of all, it allowed us to detect unforeseen issues with the language syntax and its semantics, which were both revised in consequence. For instance, property nesting relies on the fact that inner properties can only access events bounded in outer properties, and some language constructs broke this rule.

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

2. Due to its focus on simplicity, the interpreter was also used to try out several possible implementations for property satisfaction. The next subsection details them and our choices.
3. It allowed us to experiment with the selected representative properties (Chapter 3) on real surgery traces, and work with Blue Ortho engineers to diagnose issues they had at the time. Those experiments helped to validate the language fitness. Section 8.2.2 presents some of them.
4. Finally, the Command-Line Interpreter also became a reference implementation of ParTraP, which was useful in the development of the ParTraP IDE, presented later in this chapter.

8.2.1 *Implementation Strategies*

Since we are performing only offline verification in the MODMED project, we consider complete execution traces and not a prefix of them as in online monitoring. Having access to a full trace allows exploring it back and forth and not just sequentially. A direct consequence is that it is possible to interpret properties naively, in the sense that an interpreter may follow their syntactic structure. For instance, to check the property

```
before first A, P
```

the interpreter may simply process the input trace forward, looking for an event `A`, and then process the trace until `A` again to check for `P`. Although this approach is very simple, it suffers from a major source of inefficiency: it may require going through a trace interval several times (possibly many times), such as the interval before `A` in the previous example.

Of course, even if we have access to the full trace, we could process it sequentially as commonly done for online runtime verification. This requires to rewrite properties into some form of automata that will accept or not a trace. This transformation is highly non-trivial, especially when the language supports past and future operators, and parametric events. Besides the significant increase in complexity of the implementation, this approach also suffers from two important sources of inefficiency. First, it must memorize all the information that *might* be needed in the future, as the trace can only be accessed sequentially. Second, it must assume that any encountered event complying with the current state of the evaluation may become relevant in the future, and will possibly violate the property. Many of these events are actually later filtered out and monitoring for them was a waste. In fact, it was even shown that parametric online runtime verification is NP-complete [19].

To summarize, the offline strategy has a straightforward implementation but it might process the same trace interval over and over, while the online strategy has a complex implementation and must perform a lot of bookkeeping but it only goes through a trace once and supports non-finite traces. Because both approaches can be inefficient, and because the traces studied in the MODMED project are small (about 3000 events on average), implementation complexity was the decisive factor. That is why the interpreter uses the offline approach. It also turned out that error reporting is much more precise with the offline strategy, mostly because it follows the structure of the property, whereas the property is completely rewritten in the case of the online strategy.

8.2.2 *Experiments*

We used the ParTraP interpreter to verify 8 properties out of the 15 representative ones over a corpus of 100 surgery traces provided by Blue Ortho. Those properties encode requirements for TKA, hypotheses made on the device usage or usage queries. The other 7 properties were left out as the necessary information was not present in TKA traces. No errors were found in TKA software, but we discovered abnormal events and suspicious behaviors in the usage of the system. For instance, the periodically reported temperature of the 3D camera occasionally reaches the extreme value of $-273\,^{\circ}\mathrm{C}$. It is the result of a failure when querying the temperature sensor. Although not critical, this defect illustrates an issue between the software and its execution environment.

By using a temporal usage query, we also noticed that for 11 % of the traces in the corpus, the user performs an action within less than 100 ms after a new screen is displayed. Even experienced users could miss information in such a short delay. Blue Ortho was already aware of the problem but its frequency was still a surprise for them. As an experiment with ParTraP, we scheduled a meeting with a Blue Ortho engineer and used the Command-Line Interpreter to diagnose this issue. After several refinements of a property encoding the behavior of interest, and filtering out irrelevant traces, the issue was attributed to a defect in the pointer device or a poor handling of that same device. This defect is not critical because the surgeon may always go back to the missed screen. Although the language itself appeared appropriate for solving the task at hand, its usability was impaired by the rudimentary Command-Line Interpreter.

8.3 USER ENVIRONMENT

The ParTraP IDE is a toolset designed to edit and execute the ParTraP language directly on a set of trace files. It was developed to

make the installation and use of ParTraP as simple as possible. It includes a syntax-directed editor and generates detailed reports on the satisfaction of a set of properties by a set of traces, with explanations on the error causes in case of violation. On the ParTraP website[2], you may find a companion video demonstrating the tool, links to reference documents describing the syntax and semantics of ParTraP, and instructions on how to download the ParTraP IDE. The companion video illustrates the main constructs of the language and shows how the tool helps to edit properties, generates Java monitors, evaluates properties and explains the result of their evaluation.

ParTraP IDE relies on the Eclipse IDE and the Xtext language workbench [29]. Xtext provides a complete infrastructure for implementing languages, including a parser, a lexer, a typechecker and a compiler generator. Figure 8.2 shows the ParTraP IDE architecture. Part A represents how Xtext generates the toolset, while Part B depicts the process of using the tool.

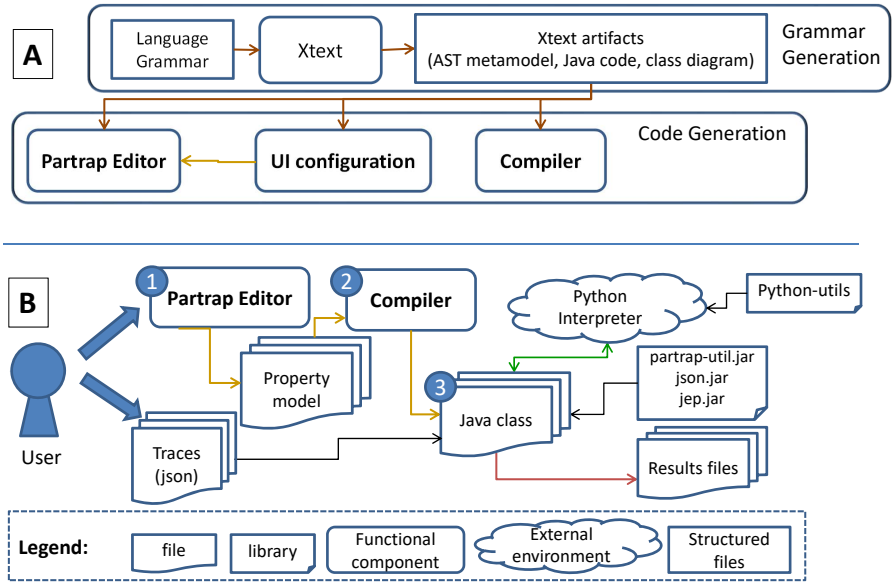


Figure 8.2: Architecture of the ParTraP toolset

8.3.1 *Tool Generation*

Part A of Figure 8.2 depicts the process responsible for generating ParTraP IDE. The ParTraP language grammar is defined with Xtext's default grammar language, a dialect of EBNF. After being parsed, a set of language models is generated (Abstract Syntax Tree, Java code and class diagram). These Xtext artifacts are used to configure the language editor and to generate a compiler that transforms each ParTraP property to a Java monitor. When a large set of properties is considered, ParTraP IDE allows to compute the whole set of properties

2 http://vasco.imag.fr/tools/partrap/

at the same time. It is less time consuming than executing separate Java classes for each property.

8.3.2 *Integrated Development Environment*

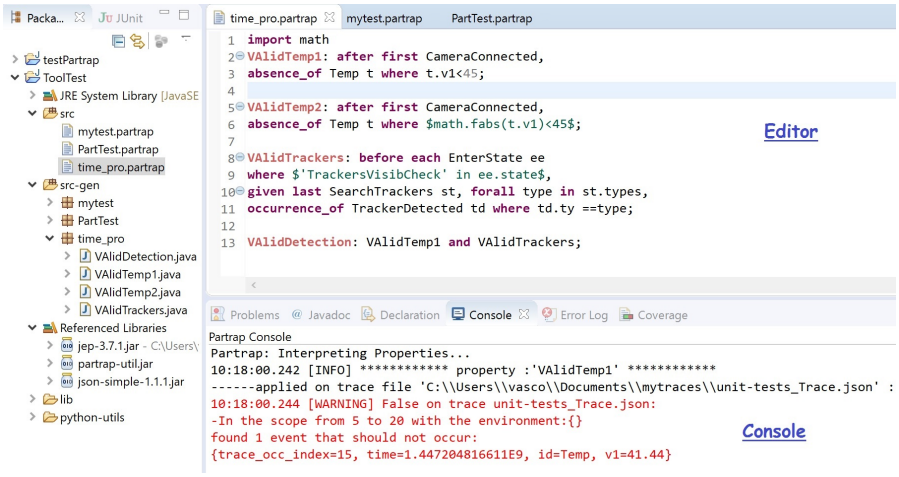


Figure 8.3: Screen capture of the environment

The architecture of the user-facing part of the IDE is depicted in Part B of Figure 8.2.

THE PARTRAP EDITOR The editor helps users to write syntactically correct properties. The configured editor provides syntax coloring according to concepts (name, keyword, python script,..) as shown in Figure 8.3 under the editor window. Python expressions are delimited by dollars signs ('$') as featured by property `VAlidTemp2` in former figure. Moreover, some validation constraints are enforced by the editor in order to forbid undesired language expressions like double use of property names or recursivity when referencing properties. Saving the file automatically calls the PARTRAP compiler and produces the set of Java classes under package `src-gen` (see project explorer in Figure 8.3).

EXECUTION AND RESULTS. Execution takes two forms: running individual java classes or evaluating all properties simultaneously. The user provides a set of traces to be evaluated. Executing the property produces a set of results files. Short logs only display the result (true or false) and an explanation for false cases. Detailed logs give information on calculated scopes, patterns and expressions results. A summary of valid and invalid traces is provided for each property.

The first property in the editor shown in Figure 8.3, named `VAlidTemp1`, encodes the requirement that the temperature should not go below 45 °C after the camera is connected. After evaluation on an artificial trace, the console at the bottom reports that one event having a temperature value of `41.44` violates this property.

Table 8.1: Evaluation times for different properties (in seconds)

Property	Native expr.	Python expr.
1	0.307	2.988
2	0.326	2.378
3	n/a	1.736
6	17.605	51.639
10	0.674	2.170
13	1.517	4.969

PYTHON EXPRESSIONS Property `VAlidTemp2` is an alternate expression of the same property whose `where` clause is expressed in Python and uses the Python `math` module. As it is important for our envisioned users to define complex calculations in property expressions, ParTraP properties support the integration of Python expressions using declared Python libraries. As a consequence, the designed IDE allows the import of Python modules and the execution of Python scripts. This is made possible by the use of JEP (Java Embedded Python)[3] which is a Python package generating a jar file 'jep.jar' added in the Java build path to exchange data and scripts between the JVM and the Python Interpreter.

PERFORMANCE Although we traded performance off against expressiveness of the language, we carried out several experiments to check that the generated monitors featured sufficient performance in the context of our industrial partner, who typically collects and analyzes several dozens of traces every day. Therefore, we collected 100 traces from our partner. After extracting events of interest, the traces range from 304 to 1163 events, with an average of 530 events. We evaluated 6 properties from the ones listed in Chapter 3. These properties typically combine a scope with a temporal pattern. For each property, we constructed a variant whose `where` clause is expressed in Python (except property 3 which already has its assertion written in Python). We conducted the experiments on a Microsoft Windows 10 machine with an Intel Core i7-6600U CPU @ 2.60GHz, and 16 GB of RAM. Each experiment was performed 50 times and the average execution times are reported in table 8.1.

These traces are not publicly available for confidentiality reasons.

Column 2 reports the time in milliseconds to evaluate the property on 100 traces. The 100 traces are covered in about a second for all properties but property 6. The significantly longer execution time for property 6 can be attributed to the fact that it exploit the `EnterState` event, which has the highest number of occurrence in the corpus.

3 `https://github.com/ninia/jep`

Moreover, the IDE implementation does not leverage the monotony of time for further optimization yet. Column 3 reports on the same properties but with their `where` clause expressed in Python. As expected, their evaluation is 4 to 10 times slower due to the extra cost of interaction between the java monitor and the Python interpreter.

These experiments show that ParTraP monitors perform well enough on traces provided by our partner. Most results are computed in just a few seconds over a 100-traces corpus, even if they involve Python assertions. Although not competitive with some of modern runtime verification approaches, these performances match the needs of our industrial use case where several dozens of traces should be processed every day.

In summary, ParTraP IDE assists the process of writing properties thanks to its syntax-directed editor. It compiles properties into monitors that can be executed on a corpus of traces with reasonable performances, and provides both synthetic and detailed reports. Moreover, the inter-operability with Python enriches ParTraP with a full fledged programming language. However, it does not help to write "correct" properties, which is especially delicate for temporal properties. For this purpose, we designed a coverage measurement tool for ParTraP properties.

9

COVERAGE MEASUREMENT FOR PARTRAP

ParTraP emphasizes readability of properties by featuring the intuitive patterns of Dwyer et al. [28], and by adopting a user-friendly syntax. However, correctly specifying properties remains a delicate task and special cases might be missed. In particular, it is easy to overlook vacuous truth: a conditional statement with a false antecedent. For instance, the propositional formula $P \Rightarrow Q$ is *vacuously true* when P is false. In ParTraP, vacuous truth may be induced by the absence of given events. Consider a property of the form

```
after first A, occurrence_of B
```

If this property is always satisfied when evaluated on a trace corpus, one may be led to believe that the temporal relation between `A` and `B` is respected. However, one should remember that `after first` is weak, i.e. it will be satisfied if `A` does not occur, regardless of `B`. In that case, the whole property will be vacuously true. Vacuous truth may induce a false sense of confidence in a system. For this reason, it was important to provide coverage information for users to detect those cases.

Measuring coverage highlights which "parts" of a property led to the satisfaction of the property. For instance, when evaluating the previous example property on a satisfying trace, knowing whether `occurrence_of B` is used reveals which scenario is encountered. Indeed, only the vacuous case can be satisfied without using `occurrence_of B`. The definition of a "part" of property depends on the approach to coverage measurement.

Coverage measurement for a declarative language such as ParTraP is not as well-established as for executable code. We considered several techniques:

- Compile properties into automata and measure their branch/state coverage. Castillos et al. studied this idea for measuring coverage of temporal properties based on Dwyer's patterns [17].
- Compile properties to an executable language for which mature code coverage tools exist, such as Java.
- Decompose properties into sub-properties, revealing the different ways to satisfy the initial property. This approach was explored for LTL by Li et al. [53].

We dismissed the first two options, based on property compilation, because relating coverage information about the compiled form to the initial property can be very difficult, and may lead to approximate results. On the other hand, property decomposition does not require this step as the formalism remains the same. As a consequence, the decomposed form can be directly shown to users so that they discover the possible sub-cases, and they can directly analyze the coverage results. For this reason, we chose the third approach.

In this chapter we first detail this decomposition technique, then describe a formal Term Rewriting System (TRS) performing this decomposition, and finally illustrate it with several examples.

9.1 MEASURING COVERAGE BY CASE DISJUNCTION

Many PARTRAP properties can be satisfied by several scenarios. The previously mentioned case of

```
after first A, occurrence_of B
```

is such an example: either `A` does not occur, or it does and it is followed by `B`. The number of possible scenarios can become much larger than 2 when properties are nested.

In order to measure coverage of a property, we propose to decompose it into sub-properties corresponding to different scenarios, exhibiting the various ways to satisfy the property. More precisely, we want to transform a property P into a disjunction of sub-properties of the form

$$P_1 \texttt{ or } P_2 \texttt{ or ... or } P_n$$

such that both forms are equivalent, and that each property from P_1 to P_n specifies a particular scenario. In the remaining of this chapter, we will say that a PARTRAP property of the above form is in Disjunctive Normal Form (DNF). Note that this form differs from the well-known DNF of propositional logic in several ways:

1. sub-properties do not have to be disjoint,
2. sub-properties may still contain disjunctions, and
3. PARTRAP's DNF is not canonical.

A coverage criterion emerges naturally for properties in DNF: we can count the number of traces in a given set that satisfy each sub-property. We say that a sub-property was covered by a trace set if its counter is positive.

The conversion of arbitrary PARTRAP properties into DNF can be achieved by

- splitting certain operators, or pairs of operator, into two properties based on the occurrence or absence of an event referenced in them, and
- pulling disjunctions outward in order to exhibit a disjunction of sub-properties at the top level.

This strategy is encoded in a TRS.

9.2 TERM REWRITING SYSTEM

In this section we define a TRS for converting ParTraP formulas into DNF. We follow the conventions for TRSs and denote rewrite rules as $P \to P'$, meaning that a term matching P is be rewritten into P'. Rules may match a whole formula or any sub-term in it, in which case only the sub-term is rewritten. Note that our TRS preserves the semantics of properties. A sufficient condition for that is to make sure that each individual rewrite rule is semantics preserving. In other words, properties must only be rewritten into equivalent ones. We say that two properties P and P' are equivalent, written $P \equiv P'$, if and only if they are satisfied by the same traces:

$$\forall P, P'((P \equiv P') \iff \forall \tau, \eta(\tau \vDash_\eta P \iff \tau \vDash_\eta P')).$$

Provided with this notation for equivalence, we can formalize the requirement that rewrite rules are semantic preserving:

$$\forall P, P'((P \to P') \implies (P \equiv P')). \tag{9.1}$$

9.2.1 *Core Operators*

It is usually a good idea to keep the number of rewriting rules small in a TRS so that it can be verified more easily. This can be achieved by minimizing the number of language operators that are used in rewriting rules. We call them *core operators*. They are individually "terminals", in the sense that they can only be rewritten when paired together. Properties using non-core operators are rewritten into equivalent ones using only core operators. The set of operators used to define the semantics of ParTraP in Section 7.3.3 is an obvious candidate for the core operators. However, a better set can be designed for the purpose of this particular TRS whose goal is to exhibit a disjunction of scenarios. In particular, this set includes new operators.

Remember that scopes in ParTraP are weak, i.e. in case where the event delimiting a scope does not occur, the whole scope is considered to be satisfied. The "`given`" operator follows the same logic. In order to distinguish the absence and the occurrence of the delimiting event, we introduce strong variants for `after`, `before` and `given`:

- `after1` *occ* A, $P =$ `occurrence_of` A `and after` *occ* A, P

- `before1` *occ* A, P = `occurrence_of` A `and before` *occ* A, P
- `given1` *occ* A, P = `occurrence_of` A `and given` *occ* A, P

These variants are suffixed with a "1" to emphasize that the delimiting event must occur at least once. The two scenarios covered by the `after` scope can now be made explicit with a simple disjunction:

$$\texttt{after } \textit{occ } A, P \equiv \texttt{absence\_of } A \texttt{ or after1 } \textit{occ } A, P.$$

The proof of this equivalence is rather direct after expanding the definition of `after1`. Similar equivalences also hold for `before` and `given` with their respective strong variants.

With these strong variants, we can now give the full set of core operators for our TRS: `or`, `not`, `occurrence_of`, `exists`, `after1`, `before1` and `given1`. All the other constructs in PARTRAP can be expressed in terms of those.

9.2.2 *Rewrite Rules*

DEFINITION EXPANSION As mentioned already, the TRS should expand all the language operators that are not core operators. The following rules simply rewrite them according to their definitions.

- P_1 `and` $P_2 \rightarrow$ `not (not` P_1 `or not` P_2`)`
- P_1 `implies` $P_2 \rightarrow$ `not` P_1 `or` P_2
- P_1 `equiv` $P_2 \rightarrow$ (`(`P_1 `implies` P_2`) and (`P_2 `implies` P_1`)`
- `forall` x `in` e, $P \rightarrow$ `not exists` x `in` e, `not` P
- `absence_of` $E \rightarrow$ `not occurrence_of` E
- A `followed_by` $B \rightarrow$ `after each` A, `occurrence_of` B
- A `preceded_by` $B \rightarrow$ `before each` A, `occurrence_of` B
- A `prevents` $B \rightarrow$ `after each` A, `absence_of` B
- `between` A `and` B, $P \rightarrow$ `after each` A, `before first` B, P
- `since` A `until` B, $P \rightarrow$ `(between` A `and` B, P`) and (`
  `after1 last` B, `after each` A, P
  `or (absence_of` B `and after each` A, P`)`
  `)`

Note that the right-hand side of some rules contains non-core operators. They will be eliminated by subsequent rewrites.

CASE DISTINCTION FOR WEAK OPERATORS We already discussed how weak scope operators can be satisfied by two kinds of scenarios. The following rules decompose them to exhibit these scenarios:

- `after` *occ* *A*, *P* → `absence_of` *A* `or after1` *occ* *A*, *P*
- `before` *occ* *A*, *P* → `absence_of` *A* `or before1` *occ* *A*, *P*
- `given` *occ* *A*, *P* → `absence_of` *A* `or given1` *occ* *A*, *P*

Weak scope operators were the last remaining non-core operators, which are now eliminated as well.

DISTRIBUTIVITY OVER DISJUNCTION Distributing certain operators over disjunctions is useful to exhibit a disjunctive form at the top-level of a formula. Conceptually, disjunctions are pulled outward and other operators are pushed inward. In ParTraP, two types of operators distribute over disjunctions: existential quantifiers, as in first-order logic, and more interestingly, the `first` and `last` variants of strong scopes. The distributivity of the existential quantifier results directly from the fact that ParTraP follows first-order logic for its non-temporal operators. The distributivity of strong scopes is justified by the fact that its `first` and `last` variants are akin to an existential quantifier, with the additional restriction on trace range. Indeed, such scope requires at least one event of a given type to occur ("there exists"), and that a given property holds for that particular event occurrence ("such that"). Note that this property does not hold for the `each` variant.

These properties on distributivity are used to rewrite existential quantifiers:

- `exists` *x* `in` *e*, (*P*$_1$ `or` *P*$_2$) →
  (`exists` *x* `in` *e*, *P*$_1$) `or` (`exists` *x* `in` *e*, *P*$_2$)

as well as strong scopes:

- `after1 first` *A*, (*P*$_1$ `or` *P*$_2$) →
  (`after1 first` *A*, *P*$_1$) `or` (`after1 first` *A*, *P*$_2$)
- `after1 last` *A*, (*P*$_1$ `or` *P*$_2$) →
  (`after1 last` *A*, *P*$_1$) `or` (`after1 last` *A*, *P*$_2$)

The rules for `before1` and `given1` are similar and omitted here.

NEGATION OF STRONG SCOPES The negation of a strong scope operator with the `first` and `last` variants is the last expression type that can be decomposed into a disjunction of two scenarios. Intuitively, a property of the form

`after1 first` *A*, *P*

can be violated by two scenarios: either *A* does not occur, or *A* does occur but *P* is not satisfied after this occurrence. This reasoning holds for the 6 possible combinations of `after1`, `before1` or `given1` together

with `first` or `last`. This decomposition is encoded by the following rewrite rules:

- `not after1 first` A, $P \to$
  `absence_of` A `or after1 first` A, `not` P
- `not after1 last` A, $P \to$
  `absence_of` A `or after1 last` A, `not` P

Once again, rules for `before1` and `given1` are similar and omitted.

NEGATION ELIMINATION Finally, many negations may appear in a formula during rewriting. In particular, double negations can prevent certain of the previous rules from firing. As in classical logic, they can be simply eliminated:

- `not not` $P \to P$

9.2.3 *Implementation*

This TRS is implemented in the publicly available Command-Line Interpreter for ParTraP[1]. It relies heavily on the pattern-matching feature of Haskell to encode the rewriting rules. A step function tries to match the top-level structure of a property and to rewrite them accordingly. The fact that sub-terms may also be rewritten is handled by applying the step function recursively if no rewrite were possible at the top-level. To fully rewrite a property, the step function is applied repeatedly until it can no longer rewrite anything. In other words, we compute a fixed point for the step function. Once a property is fully rewritten, its top-level disjunctions are broken down to produce a list of sub-properties. The coverage level of each of these sub-properties is computed by evaluating them separately over a trace set and counting the number of times they are satisfied.

9.3 COVERAGE EXAMPLES: TKA PROPERTIES

We now illustrate the usage of coverage measurement based on DNF decomposition on several examples extracted from the TKA case study presented in Chapter 2. Evaluation was performed on a sample of 100 surgery traces.

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

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

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:

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

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:

(a) `absence_of EnableTracker enable`

(b) `before1 each EnableTracker enable, occurrence_of TrackerDetected detect where detect.id == enable.id`

Sub-property (a) captures the potential absence of `EnableTracker` whereas (b) uses the stricter version `before1` that requires at least one occurrence of this event. 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: 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 technicals 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) `after1 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 a misuse 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 Blue Ortho (dealing with shoulder surgery). It is only through coverage measurement that we noticed that the provided trace corpus was heterogeneous.

Example 3: Complex Temporal Property

As a last example, consider Property 12 and its ParTraP expression given in Section 6.3:

```
since ActionPrevious until AcquisitionBegin,
    absence_of AcquisitionSuccess
```

Although this property looks simple, the `since/until` scope actually captures many possible scenarios. Indeed, this property is rewritten in DNF as the disjunction of 8 sub-properties. For conciseness, we give this list with `ActionPrevious`, `AcquisitionBegin`, and `AcquisitionSuccess` shortened as `AP`, `AB`, and `AS`, respectively:

(a) `absence_of AP and absence_of AB`

(b) `absence_of AP and absence_of AB and after1 each AP, absence_of AS`

(c) `absence_of AP and after1 last AB, absence_of AP`

(d) `absence_of AP and after1 last AB, after1 each AP, absence_of AS`

(e) `after1 each AP, (absence_of AB or before1 first AB, absence_of AS) and after1 last AB, absence_of AP`

(f) `after1 each AP, (absence_of AB or before1 first AB, absence_of AS) and after1 last AB, after1 each AP, absence_of AS`

(g) `after1 each AP, (absence_of AB or before1 first AB, absence_of AS) and absence_of AB and absence_of AP`

(h) `after1 each AP, (absence_of AB or before1 first AB, absence_of AS) and absence_of AB and after1 each AP, absence_of AS`

Sub-property (c) is covered by 30 traces, (e) by 45 traces, and (f) by 25 traces. Together they account for the 100 traces of the corpus. The remaining cases are not covered. This coverage information for TKA is not particularly interesting to discuss. Instead, we will focus on the decomposition itself.

Inspecting the different sub-properties reveals two limitations of using a TRS to perform DNF decomposition. First, it is unable to simplify some sub-properties, which impairs their readability. This can be observed in sub-property (c), where `absence_of AP` inside the scope is redundant with the same constraint at the top-level. Second, and most importantly, the TRS sometimes introduces unsatisfiable sub-properties. Those undesirable additions might confuse users. In the previous decomposition, sub-properties (b), (d) and (g) are all unsatisfiable due to a contradiction on the absence and occurrence of the event `ActionPrevious`. Note that the decomposition remains correct since introducing unsatisfiable propositions in a disjunction does not change its semantics. Both limitations are due to the fact that the TRS operates at a purely syntactic level.

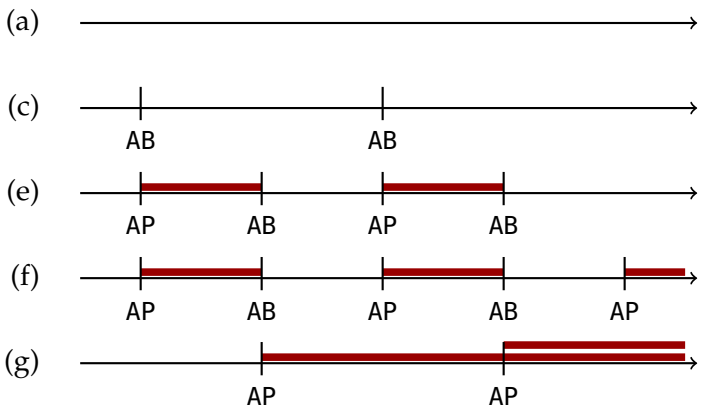


Figure 9.1: Graphical representation of the 5 valid scenarios in Property 12

Looking past those limitations, one can observe that the decomposed form exhibits 5 different scenarios, described by the 5 satisfiable sub-properties. There are graphically represented with timelines in Figure 9.1. Colored intervals represent ranges where `AcquisitionSuccess` must not occur. While our TRS was only differentiating vacuity from non-vacuity in the examples of the two previous sections, the present decomposition with various scenarios shows that this approach also supports richer temporal properties.

9.4 FURTHER DECOMPOSITION

Despite its limitations (some properties could be simplified and others are unsatisfiable), coverage measurement based on DNF decomposition does lead to better insights on properties and the trace sets they

are evaluated on. This decomposition technique attempts 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.

Consider a property of the form

`after each` A`,` P_1 `or` P_2.

It can be decomposed according to the number of event occurrences matching the event descriptor 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 {1,1}` A `and (after first` A`,` P_1 `or` P_2`)`

(c) `occurrence_of {2,2}` 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`)`

The notation $\{a,b\}$ is the same as the one used in POSIX regular expressions. It indicates that an event must occur at least a times and at most b times. As in regular expressions, it is possible to specify only one of the two bounds. The above disjunction can be further rewritten to exhibit several cases at the top level:

(a) `absence_of` A

(b) `occurrence_of {1,1}` A `and (after first` A`,` P_1`)`

(c) `occurrence_of {1,1}` A `and (after first` A`,` P_2`)`

(d) `occurrence_of {2,2}` A `and (after first` A`,` P_1`) and (after last` A`,` P_1`)`

(e) `occurrence_of {2,2}` A `and (after first` A`,` P_1`) and (after last` A`,` P_2`)`

(f) `occurrence_of {2,2}` A `and (after first` A`,` P_2`) and (after last` A`,` P_1`)`

(g) `occurrence_of {2,2}` 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.type == 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 before1 first StartAcquisition,` P`)`, where

$P =$ `forall ty in st.types,`
`occurrence_of TrackerDetected td where td.type == 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 {1,1} SearchTrackers and (after first SearchTrackers st, absence_of StartAcquisition)`

(c) `occurrence_of {1,1} SearchTrackers and (after first SearchTrackers st, before1 first StartAcquisition,` P`)`

(f) `occurrence_of {2,2} SearchTrackers and (after first SearchTrackers st, before1 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. A closer inspection reveals that the two traces satisfying this case were actually generated by the shoulder product, mentionned 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.

CONCLUSION In spite of the encouraging results, this second decomposition level was only partially formalized and partially implemented due to time constraints. On the contrary, the first level of decomposition based on the absence of occurrence of given events is fully implemented. We used this implementation to analyze the coverage of several TKA properties. Those experiments confirmed that coverage measurement based on DNF decomposition helps to better understand temporal properties, the results of their evaluation on a corpus of traces, and to identify faulty or incomplete properties. They also highlighted the main limitation the approach based on TRS: due to its purely syntactic nature, the TRS is not able to simplify some properties. Thus, it produces longer sub-properties than necessary, and sometimes introduces sub-properties that are not satisfiable.

10

CONCLUSION AND PERSPECTIVES

10.1 SUMMARY

This PhD research was conducted as part of the MODMED project, a public research project gathering a laboratory and two companies. Blue Ortho manufactures medical devices which provide guidance during complex surgeries, such as total knee arthroplasty, and MinMaxMedical develops software components for those devices. In this PhD thesis, I tackled the main goal of the MODMED project: developing a specification language for verifying temporal properties on data-rich execution traces, and most importantly, tailored for industrial usage.

First of all, we conducted a thorough study of TKA, the main product commercialized by Blue Ortho. This surgery assistant for total knee arthroplasty appears as representative of the new generation of medical devices, and has been used worldwide for several years. Based on several interviews with Blue Ortho, the specification documents for TKA, and multiple execution traces recorded during real surgeries, we extracted a list of properties that are representative of the properties that manufacturers would verify on execution traces. Those properties were temporal for the most part, relied heavily on data carried by events in execution traces, and required exploiting data with complex structure. Comparing many existing formalisms for temporal property specification with respect to those characteristics revealed that none of them was suitable. Thus, we set out to design a solution that would meet those criteria while being accessible to engineers with no training in formal methods.

Our solution is called ParTraP. It is a specification language for parametric execution traces that is designed to be simple to use by software engineers thanks to a declarative style and a user-friendly syntax. It allows formalizing properties involving temporal constraints and data values concisely, while remaining readable. ParTraP was inspired by specification patterns and extends them with nested scopes, real-time and first-order quantification. Besides its user-friendly syntax, the language relies on simplicity to remain accessible: it requires understanding only a few concepts, which can be applied to most of the language constructs. In particular, all properties can be freely nested, and all event descriptors can be associated to a name and constrained. The combination of these two simple mechanisms allows

writing a variety of temporal properties, including properties that relates several event occurrences according to their parameters.

To be ready for real-world use, ParTraP comes with an IDE with binary releases and sources available under the GNU Lesser General Public License (LGPL). It packages a syntax-directed editor, assisting the process of writing properties, a compiler, and an execution environment providing both synthetic and detailed reports. Moreover, the inter-operability with Python enriches ParTraP with a full fledged programming language. While execution performance is modest and not competitive with state-of-art runtime verification techniques, experiments showed that ParTraP monitors perform well enough on traces provided by Blue Ortho.

Finally, despite our best efforts to make ParTraP simple to use, writing temporal specification remains a delicate task. Some properties that may seem easy to understand can divert the reader from careful consideration of special cases. 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). 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 we devised and implemented is able to execute the first level of decomposition. Unfortunately, it also introduces tautologies or contradictions in the decomposed form. While the result remain correct, those undesirable additions might confuse users. Nonetheless, we showed its usefulness through several examples extracted the TKA 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.

10.2 FUTURE RESEARCH DIRECTIONS

Based on the current state of ParTraP, we describe five research directions that would consolidate the language and its tool set.

Usability Study

Although we believe that ParTraP makes parametric property specification approachable by engineers unqualified with formal methods, we did not conduct a full study to back-up this claim. As of this day, we only gathered anecdotal evidence of ParTraP suitability to its task: the language was demonstrated to several engineers from MinMaxMedical and Blue Ortho who validated its design. Furthermore,

ParTraP was used successfully by a dozen of people, ranging from graduate students to full-time researchers, to perform trace analysis and verification on TKA traces as well as traces from other projects. Nonetheless, conducting a usability study of ParTraP with a monitored hands-on session remains a future work.

Application to Other Contexts

ParTraP's design was driven by logs and requirements from the medical field. Yet, it is not specific to that particular field. The generation and gathering of execution traces is already a standard practice in many industries, but their systematic and automatic exploitation is not generalized. ParTraP is transposable to most of them, and would be particularly useful for traces with complex structured data. Our research group is currently working on a case study in the field of home automation, where traces include parametric information from sensors, and actions performed by the inhabitants and by the control system. At longer term, we have contacts with another medical device manufacturer that we met through our industrial partners. The growing complexity of their execution traces is becoming a concern, which ParTraP could alleviate.

Better Property Decomposition

There has been little research on coverage measurement for declarative languages compared to procedural languages. We showed how decomposing ParTraP properties in Disjunctive Normal Form can provide useful coverage information, helping to understand both properties and traces on which they are evaluated. However, only the first level of decomposition, based on absence or occurrence of certain events, was formalized and implemented. For certain types of properties, this decomposition is insufficient and does not provide insightful results. The second decomposition level relying on the number of occurrences of certain events is promising in this situation, and worth pursuing. Furthermore, the major limitation of this approach (i.e. the production of tautologies and contradictions) is due to the syntactic nature of the term rewriting system. Overcoming this limitation would greatly improve the reliability of property decomposition, and is an interesting research perspective.

Example and Counter-Example Generation

Generating examples and counter-examples for properties, i.e. satisfying and violating traces, would further help users to understand the properties they write, and to assess their correctness. To complement ParTraP's tool set, the development of such a tool was started recently by our research group. The current prototype relies on the modern Satisfiability Modulo Theories (SMT) solver Z3 [57], which is

able to provide a model for a formula. In this case, the model is the trace example, while the formula is the encoding of a property. However, a single model, or trace, is produced. As a consequence, the examples produced by the current prototype only demonstrate one of the many possible scenarios encoded in a temporal property. We believe this limitation could be tackled by combining our DNF decomposition technique, which exhibits the different temporal scenarios behind a property, with the current SMT based solution.

Perfomance Improvements

Finally, current ParTraP implementations (Command-Line Interpreter and compiled monitors) only have modest performance, and it is probably possible to improve them. In the context of MODMED, we studied complex and rich, but relatively small execution traces. Therefore, our focus has been on the language design and its tooling, and less on the efficiency of its implementation. While we validated the ability of our implementations to process longer traces synthetically generated, we are not sure that these traces are representative of the ones that can be found in other contexts. Since the current compiler in ParTraP IDE does not perform any form of optimization, we expect that producing optimized monitors would lead to substantial performance improvements.

BIBLIOGRAPHY

[1] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. "Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors." In: *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. Lecture Notes in Computer Science. Springer, 2012, pp. 68–84. DOI: 10.1007/978-3-642-32759-9_9.

[2] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. "Rule-Based Runtime Verification." In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. 2004, pp. 44–57. DOI: 10.1007/978-3-540-24622-0_5.

[3] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret H. Smith. "Formal Analysis of Log Files." In: *JACIC* 7.11 (2010), pp. 365–390. DOI: 10.2514/1.49356.

[4] Howard Barringer, David E. Rydeheard, and Klaus Havelund. "Rule Systems for Run-time Monitoring: from Eagle to RuleR." In: *J. Log. Comput.* 20.3 (2010), pp. 675–706. DOI: 10.1093/logcom/exn076.

[5] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. "Monitoring of temporal first-order properties with aggregations." In: *Formal Methods in System Design* 46.3 (2015), pp. 262–285. DOI: 10.1007/s10703-015-0222-7.

[6] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. "Monitoring Metric First-Order Temporal Properties." In: *J. ACM* 62.2 (2015), p. 15. DOI: 10.1145/2699444.

[7] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Comparing LTL Semantics for Runtime Verification." In: *J. Log. Comput.* 20.3 (2010), pp. 651–674. DOI: 10.1093/logcom/exn075.

[8] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL." In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), p. 14. DOI: 10.1145/2000799.2000800.

[9] Andreas Bauer, Martin Leucker, and Jonathan Streit. "SALT - Structured Assertion Language for Temporal Logic." In: *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*. Ed. by Zhiming Liu and Jifeng He.

Vol. 4260. Lecture Notes in Computer Science. Springer, 2006, pp. 757–775. DOI: 10.1007/11901433_41.

[10] Andrew R. Bernat and Barton P. Miller. "Anywhere, any-time binary instrumentation." In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11, Szeged, Hungary, September 5-9, 2011*. Ed. by Jeff Foster and Lori L. Pollock. ACM, 2011, pp. 9–16. DOI: 10.1145/2024569.2024572.

[11] A Blandford, G Buchanan, P Curzon, D Furniss, and H Thimbleby. "Who's looking? Invisible problems with interactive medical devices." In: *Proceedings of the First International Workshop on Interactive Systems in Healthcare*. ACM Special Interest Group on Computer-Human Interaction. 2010, pp. 9–12.

[12] 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*. Tech. rep. LIG, MinMaxMedical, BlueOrtho, 2017.

[13] Eric Bodden and Klaus Havelund. "Racer: effective race detection using AspectJ." In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. Ed. by Barbara G. Ryder and Andreas Zeller. ACM, 2008, pp. 155–166. DOI: 10.1145/1390630.1390650.

[14] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. "Sampling-Based Runtime Verification." In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 88–102. DOI: 10.1007/978-3-642-21437-0_9.

[15] Ricky W Butler, James L Caldwell, Victor A Carreno, C Michael Holloway, Paul S Miner, and Ben L Di Vito. "NASA Langley's research and technology-transfer program in formal methods." In: *Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*. IEEE. 1995, pp. 135–149.

[16] David A. Bversiooasin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. "MONPOLY: Monitoring Usage-Control Policies." In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 360–364. DOI: 10.1007/978-3-642-29860-8_27.

[17] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. "Coverage Criteria for Model-Based Testing using Property Patterns." In: *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014*. Ed. by Holger Schlingloff and Alexander K. Petrenko. Vol. 141. EPTCS. 2014, pp. 29–43. DOI: 10.4204/EPTCS.141.3.

[18] Feng Chen and Grigore Rosu. "Parametric Trace Slicing and Monitoring." In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Stefan Kowalewski and Anna Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 246–261. DOI: 10.1007/978-3-642-00768-2_23.

[19] Zhe Chen. "Parametric runtime verification is NP-complete and coNP-complete." In: *Inf. Process. Lett.* 123 (2017), pp. 14–20. DOI: 10.1016/j.ipl.2017.02.006.

[20] Arnaud Clère, Yoann Blein, and Yves Ledru. *Generic Execution Traces Specification*. Tech. rep. MinMaxMedical, LIG, 2018.

[21] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. "Expressing checkable properties of dynamic systems: the Bandera Specification Language." In: *STTT* 4.1 (2002), pp. 34–56. DOI: 10.1007/s100090200075.

[22] David Crocker. "Can C++ be made as safe as SPARK?" In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*. Ed. by Michael Feldman and S. Tucker Taft. ACM, 2014, pp. 5–12. DOI: 10.1145/2663171.2663179.

[23] Douglas Crockford. "The application/JSON Media Type for JavaScript Object Notation (JSON)." In: *RFC* 4627 (2006), pp. 1–10. DOI: 10.17487/RFC4627.

[24] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. "LOLA: Runtime Monitoring of Synchronous Systems." In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26.

[25] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. "Using contract-based component specifications for virtual integration testing and architecture design." In: *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*. IEEE, 2011, pp. 1023–1028. DOI: 10.1109/DATE.2011.5763167.

[26] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. "Runtime Monitoring with Union-Find Structures." In: *TACAS*. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 868–884. DOI: 10.1007/978-3-662-49674-9_54.

[27] Normann Decker, Martin Leucker, and Daniel Thoma. "Monitoring modulo theories." In: *STTT* 18.2 (2016), pp. 205–225. DOI: 10.1007/s10009-015-0380-3.

[28] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-State Verification." In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*. Ed. by Barry W. Boehm, David Garlan, and Jeff Kramer. ACM, 1999, pp. 411–420. DOI: 10.1145/302405.302672.

[29] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 307–309. DOI: 10.1145/1869542.1869625.

[30] Yliès Falcone and Sebastian Currea. "Weave droid: aspect-oriented programming on Android devices: fully embedded or in the cloud." In: *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. Ed. by Michael Goedicke, Tim Menzies, and Motoshi Saeki. ACM, 2012, pp. 350–353. DOI: 10.1145/2351676.2351744.

[31] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. "A Taxonomy for Classifying Runtime Verification Tools." In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 241–262. DOI: 10.1007/978-3-030-03769-7_14.

[32] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. "Runtime enforcement monitors: composition, synthesis, and enforcement abilities." In: *Formal Methods in System Design* 38.3 (2011), pp. 223–262. DOI: 10.1007/s10703-011-0114-4.

[33] Alessandro Fantechi, Wan Fokkink, and Angelo Morzenti. "Some trends in formal methods applications to railway signaling." In: *Formal Methods for Industrial Critical Systems* (2012), pp. 61–84.

[34] Sebastian Fischmeister and Patrick Lam. "On Time-Aware Instrumentation of Programs." In: *15th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2009, San Francisco, CA, USA, 13-16 April 2009*. IEEE Computer Society, 2009, pp. 305–314. DOI: 10.1109/RTAS.2009.26.

[35] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. "On the Temporal Basis of Fairness." In: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*. Ed. by Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne. ACM Press, 1980, pp. 163–173. DOI: 10.1145/567446.567462.

[36] 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*. IEEE Computer Society, 2008, pp. 63–72. DOI: 10.1109/EDOC.2008.32.

[37] Matthias Hauswirth and Trishul M. Chilimbi. "Low-overhead memory leak detection using adaptive statistical profiling." In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*. Ed. by Shubu Mukherjee and Kathryn S. McKinley. ACM, 2004, pp. 156–164. DOI: 10.1145/1024393.1024412.

[38] Klaus Havelund. "Rule-based runtime verification revisited." In: *STTT* 17.2 (2015), pp. 143–170. DOI: 10.1007/s10009-014-0309-2.

[39] Klaus Havelund and Giles Reger. "Specification of Parametric Monitors." In: *SyDe Summer School*. Springer, 2015, pp. 151–189. DOI: 10.1007/978-3-658-09994-7_6.

[40] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. "Software monitoring with controllable overhead." In: *STTT* 14.3 (2012), pp. 327–347. DOI: 10.1007/s10009-010-0184-4.

[41] Graham Hutton and Erik Meijer. "Monadic Parsing in Haskell." In: *J. Funct. Program.* 8.4 (1998), pp. 437–444. URL: http://journals.cambridge.org/action/displayAbstract?aid=44175.

[42] *ISO 13485:2016: Medical devices – Quality management systems – Requirements for regulatory purposes*. Standard. Geneva, Switzerland: International Organization for Standardization, Mar. 2016. URL: https://www.iso.org/standard/59752.html.

[43] *ISO 14971:2007: Medical devices – Application of risk management to medical devices*. Standard. Geneva, Switzerland: International Organization for Standardization, Mar. 2007. URL: https://www.iso.org/standard/38193.html.

[44] *ISO/IEC 62304:2006: Medical device software – Software life cycle processes*. Standard. Geneva, Switzerland: International Organization for Standardization, May 2006. URL: https://www.iso.org/standard/38421.html.

[45] Daniel Jackson and Jeannette Wing. "Lightweight Formal Methods." In: *ACM Comput. Surv.* 28.4 (1996), p. 121. DOI: 10.1145/242224.242380.

[46] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. "From signal temporal logic to FPGA monitors." In: *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*. IEEE, 2015, pp. 218–227. DOI: 10.1109/MEMCOD.2015.7340489.

[47] Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Nickovic. "Quantitative Monitoring of STL with Edit Distance." In: *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*. Ed. by Yliès Falcone and César Sánchez. Vol. 10012. Lecture Notes in Computer Science. Springer, 2016, pp. 201–218. DOI: 10.1007/978-3-319-46982-9_13.

[48] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. "JavaMOP: Efficient parametric runtime monitoring framework." In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 1427–1430. DOI: 10.1109/ICSE.2012.6227231.

[49] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ." In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–353. DOI: 10.1007/3-540-45337-7_18.

[50] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective." In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7.

[51] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. "PEBIL: Efficient static binary instrumentation for Linux." In: *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*. IEEE Computer Society, 2010, pp. 175–183. DOI: 10.1109/ISPASS.2010.5452024.

[52] Nancy G. Leveson and Clark Savage Turner. "Investigation of the Therac-25 Accidents." In: *IEEE Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[53] 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*. Vol. 8051. Lecture Notes in Computer Science. Springer, 2013, pp. 256–270.

[54] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals." In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*. Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

[55] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. "The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML." In: *J. Log. Algebr. Program.* 58.1-2 (2004), pp. 89–106. DOI: 10.1016/j.jlap.2003.07.006.

[56] Dominique Méry, Bernhard Schätz, and Alan Wassyng. "The Pacemaker Challenge: Developing Certifiable Medical Devices (Dagstuhl Seminar 14062)." In: *Dagstuhl Reports* 4.2 (2014), pp. 17–38. DOI: 10.4230/DagRep.4.2.17.

[57] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[58] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante

and Kathryn S. McKinley. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[59] John T. O'Donnell, Cordelia V. Hall, and Rex L. Page. *Discrete mathematics using a computer (2. ed.)* Springer, 2006. ISBN: 978-1-84628-241-6. DOI: 10.1007/1-84628-598-4.

[60] Alain OURGHANLIAN. "Evaluation of static analysis tools used to assess software important to nuclear power plant safety." In: *Nuclear Engineering and Technology*. Special Issue on ISOFIC/ISSNP2014 47.2 (2015), pp. 212–218. DOI: 10.1016/j.net.2014.12.009.

[61] Amir Pnueli. "The Temporal Logic of Programs." In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[62] Amir Pnueli and Aleksandr Zaks. "PSL Model Checking and Run-Time Verification Via Testers." In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 573–586. DOI: 10.1007/11813040_38.

[63] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. "MarQ: Monitoring at Runtime with QEA." In: *TACAS*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 596–610. DOI: 10.1007/978-3-662-46681-0_55.

[64] Giles Reger and Klaus Havelund. "What Is a Trace? A Runtime Verification Perspective." In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9953. Lecture Notes in Computer Science. 2016, pp. 339–355. DOI: 10.1007/978-3-319-47169-3_25.

[65] Giles Reger and David E. Rydeheard. "From First-order Temporal Logic to Parametric Trace Slicing." In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. 2015, pp. 216–232. DOI: 10.1007/978-3-319-23820-3_14.

[66] Germán Regis, Renzo Degiovanni, Nicolás D'Ippolito, and Nazareno Aguirre. "Specifying Event-Based Systems with a Counting Fluent Temporal Logic." In: *ICSE (1)*. IEEE Computer Society, 2015, pp. 733–743. DOI: 10.1109/ICSE.2015.86.

[67] Christoph Schnurr, Isabell Güdden, Peer Eysel, and Dietmar Pierre König. "Influence of computer navigation on TKA revision rates." In: *International orthopaedics* 36.11 (2012), pp. 2255–2260.

[68] Roger S. Scowen. *Extended BNF — A generic base standard*. Tech. rep. 1998.

[69] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. "InterAspect: aspect-oriented instrumentation with GCC." In: *Formal Methods in System Design* 41.3 (2012), pp. 295–320. DOI: 10.1007/s10703-012-0171-3.

[70] Ruben Sipos, Dmitriy Fradkin, Fabian Mörchen, and Zhuang Wang. "Log-based predictive maintenance." In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. Ed. by Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani. ACM, 2014, pp. 1867–1876. DOI: 10.1145/2623330.2623340.

[71] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. "PROPEL: an approach supporting property elucidation." In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. 2002, pp. 11–21. DOI: 10.1145/581339.581345.

[72] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. "Formal Verification of Avionics Software Products." In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Ed. by Ana Cavalcanti and Dennis Dams. Vol. 5850. Lecture Notes in Computer Science. Springer, 2009, pp. 532–546. DOI: 10.1007/978-3-642-05089-3_34.

[73] Olaf Spinczyk and Daniel Lohmann. "The design and implementation of AspectC++." In: *Knowl.-Based Syst.* 20.7 (2007), pp. 636–651. DOI: 10.1016/j.knosys.2007.05.004.

[74] Volker Stolz. "Temporal Assertions with Parametrised Propositions." In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 176–187. DOI: 10.1007/978-3-540-77395-5_15.

[75] 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." In: *Formal Asp. Comput.* 27.4 (2015), pp. 641–664. DOI: 10.1007/s00165-014-0328-5.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*.