TOWARDS AUTOMATED PERFORMANCE ANALYSIS OF PROGRAMS BY RUNTIME VERIFICATION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2021

Joshua H Dawes

Department of Computer Science

Contents

Word Count: 75,442

List of Figures

Abstract

This thesis makes a contribution to the field of Runtime Verification, a *lightweight* formal method for the analysis of computational systems. The contribution is made in multiple parts. First, a new language is introduced for the specification of properties at the source code level of programs. These properties tend to be with respect to program performance. Second, automatic monitoring and instrumentation techniques are introduced for the specification language. Third, an approach for explaining violations of these properties by program runs is introduced. Finally, the resulting body of theoretical work is implemented in an extensive ecosystem of tools for program analysis. This ecosystem is described in detail, along with its application to a real-world system at CERN.

The work presented in this thesis diverges from past work in the Runtime Verification community. Instead of focusing on maximising expressiveness of the specification formalism and solving the resulting monitoring and instrumentation problems, it focuses on introducing a language in which properties that often need to be checked over real-world programs can easily be expressed.

In the direction of instrumentation, the source-code level of abstraction of our specification language allows an approach to instrumentation that diverges from much previous work. Many previous approaches have treated instrumentation as a separate problem from specification, usually providing a language in which one can describe how instrumentation should be performed. With our specification language, instrumentation can be performed automatically with respect to a specification.

Further, an area that has received little attention in the Runtime Verification community is the analysis of verdicts resulting from monitoring programs with respect to specifications. The contributions to this area described in this thesis take the form of tools in the ecosystem. These tools enable detailed exploration of monitoring information, and mark a step towards automated generation of explanations of verdicts.

Following the description of the extensive set of tools, this thesis concludes with an in-depth discussion of their application to perform significant analyses of software used at CERN. Ultimately, the work described, including the theoretical foundations and implementations, forms the beginnings of a program analysis project whose aim, through continued development at CERN, is to enable detailed analysis of the performance of programs by software engineers with minimal effort.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First, I'll thank the people who have been involved in teaching the stuff I've needed to get my PhD.

I must first thank Giles, my Manchester supervisor, who agreed to supervise an extremely unconventional PhD and was happy to let me direct the research as I wanted. Also, whether you intended to or not, you helped me take a more relaxed (and healthy) approach to life in research.

Thanks go to my CERN supervisors, Giovanni and Andreas:

Giovanni - it's to you that I owe thanks for my immense professional development while at CERN. Your work ethic, level of organisation and way of dealing with the people around you has served as a great source of inspiration. I started my PhD when I was 22 and, as a young doctoral student learning to navigate the often overwhelming aspects of professional life at CERN, your input has been invaluable.

Andreas - I thank you profoundly for selecting me to come to CERN when I was a 20 year old undergraduate at Manchester. Coming to CERN gave me confidence, allowed me to see the world and kickstarted what I consider to be the best part of my life so far.

It may seem a little out of the blue, but I also want to thank Jo Turner, who taught me maths during my A-Levels. When I started my A-Levels, maths was a chore, and something I wanted to get out of the way. I also really struggled with it. It's thanks to Jo that I found a love for the subject that I hope I will get to work on for the rest of my life.

To the people who I've had the privilege of working with during my time at CERN.

Marta - your contributions to the VyPR project at CERN were exceptional. You have my profound thanks for all of your work on the analysis tools.

Omar - it was nice to finally have another PhD student to work with in my final year when you came to CERN. Our long discussions have shown me parts of Computer Science that I didn't know existed, and have broadened my research horizons. I will never forget mine and Sarah's visits to Lugano to see your family. May there be many more.

Now it's time to thank people who've taught me other, less-mathsy stuff.

I thank my best friends in Geneva who have managed to stay long enough in this city to see me finish my PhD.

Meghan - you were one of very few people who saw me go through some of the less fun things that have happened in the last 5 years. You were always helpful. Thank you.

Lisa - I'm tempted to just say "nug nugs!!!" but that wouldn't do justice to the happiness

you've brought me (and Sarah). Anyway, I thought that in the end you might like a mention in the thesis that speeds up Google Chrome for you.

Dimitri - je te remercie de m'avoir appris ta partie préférée de la langue française (évidemment la phrase "avoir la gueule de bois"). Quand-même, on a passé de bonnes soirées et, franchement, tu me manqueras.

Amy - I first met you at the Eurovision party at Pickwick's in Geneva. Since then, I've surprised you with my baking skills and you've made me one of the best Christmas dinners I've ever tasted. I still tell people how I commandeered all of the gravy boats. Now, when are you booking your tickets to visit me and Sarah, again?

Chris - the complete evening drinks host package: endless refills (I blame you for my accidentally finishing a bottle of limoncello), and amazing breakfasts the morning after that "one drink" at Le Chat Noir.

Finally, I thank my family.

I thank my brother, Simon - thank you for endlessly showing how proud you are of me.

I thank my dad, Geoff - thank you for letting me take up your evenings with utterly random conversation over video calls.

I thank my other half, Sarah - my partner in life, the person with whom I've shared the stresses of my PhD, and the person without whom the world would not make sense.

Finally, I thank my mum, Cheryl. Losing you those 4 years ago has left marks on me that I'm still finding. I'm trying my best to make you proud.

This thesis is dedicated to you, mum.

Chapter 1

Introduction

This thesis introduces a formal approach to the analysis of runtime behaviour of programs that is accessible to software engineers. The approach consists of 1) describing the expected behaviour of a program by writing source code level specifications over the code inside individual functions; and 2) analysing the results of monitoring with respect to such specifications. Our focus will be on specifying properties over program performance, such as timing and values held in memory. Alongside the theoretical contributions, emphasis is placed on making the new formalisms for specification and analysis accessible and intuitive for use by software engineers in the software development process. Given the new formalisms and intuitive tools provided alongside them, programs and the way in which they are used can be optimised depending on the outcome of analyses performed.

The exact motivation for understanding behaviour well varies between domains. In some, insufficient understanding can lead to loss of data or time and, in others, it can lead to loss of life. Further, the exact definition of *behaviour* can vary and is often closely tied to the domain. In this thesis, we will focus on programs employed in data-intensive domains.

Our case study is concerned with a program used in the data-intensive domain of High-Energy Physics (HEP). We will perform extensive analyses over critical software used at the CMS Experiment [Col08] at CERN.

For programs used in data-intensive domains, including HEP, it is often important to understand the timing involved in computations with respect to the data being processed. In particular, if the data being processed by the program can be divided into a discrete set of categories, it would be natural for engineers to wonder if their code works faster for certain categories. Further, it would be normal for the expected time taken by some operations to depend on the category of data being processed, so engineers' expectations would vary with the category of data.

Moving away from time taken by operations, engineers often care about things such as values held in memory and their attributes, such as type and size. Such quantities often give direct indications of behaviour of the code that generated them.

Alternatively, if it is known how a program behaves with respect to its input and, in some cases, the environment in which the program runs, then it might be possible to optimise certain

workflows that use the program depending on knowledge of how quickly it will operate.

If understanding of the behaviour of a program used in a data-intensive domain is not sufficient, data could be lost. As an example, one could consider a scenario in which some process takes too long and its input data must be thrown away due to storage restrictions. The result of such a process may be impossible to obtain without the input data if the process is ended prematurely. Hence, resolution of performance problems in this scenario is critical. Another example could be that the type of data given as input to some process is wrong and an exception is thrown without being caught; cases like this can lead to errors that lose data and are difficult to track.

The aim of the research presented in this thesis is to provide a tool that one can easily use to identify such problems that may be hard to find by existing program testing or verification techniques. Often, problems that are identified using a program analysis method can be rectified, leading to improved program behaviour. But sometimes the causes of problems are not within the reach of engineers. In these cases, we settle instead for providing methods to help better understand them and perhaps mitigate them by tuning other parts of the program.

1.1 Contributions

We give an overview of the contributions that have been made by the research in this thesis to the Runtime Verification and Software Engineering communities.

Specifying Source Code-level Behaviour. Our first contribution is the formalism described in Chapter 3 that allows software engineers to analyse the performance of their programs by writing specifications over the source code-level behaviour of their programs. This contribution was first introduced in our paper at SAC-SVT 2019 [DR19b], along with initial instrumentation and monitoring algorithms.

Explanation of Property Violations. The natural next question to ask once an implementation has been built that can check a program for satisfaction of some properties at runtime is *When a property doesn't hold, why?* Our paper at RV 2019 [DR19a] addressed this question, marking some of the first work in RV to do so. We build on that paper in Chapter 5.

Implementation. The use of the theory described in Chapter 3 to develop a prototype tool was described in our paper at TACAS 2019 [DRF+19], which also built on our previous instrumentation and monitoring work. The implementation is described in Chapter 6.

Runtime Verification in Software Engineering. We began to discuss Runtime Verification as a Software Engineering tool in our paper at CHEP 2019 (the High Energy Physics conference for Computing) [DHR+19]. Chapter 6 describes the work in this paper in much more detail.

Final Demonstration. All of the work in this thesis was presented in the form of an extended tutorial at RV 2020 [DHJ+20]. This tutorial involved a detailed discussion of the theoretical

foundations developed throughout this thesis, alongside live demonstrations of the tools presented.

Runtime Verification with Testing. A tangent of this work performed at the CMS Experiment at CERN was the addition of RV-based performance analysis into Continuous Integration processes. This is described in our paper at ASE 2020 [JDH+20] but will not be explored more in this thesis.

In the remainder of this chapter, we describe the contents of each of these publications in more detail.

1.2 Definitions of Behaviour

In order to decide on the definitions of behaviour on which we should focus, we consider the definitions of interest in our case study from Chapter 7: the Conditions Uploader [Daw17] for the CMS Experiment at CERN. Here, continuing with the theme of what can be analysed at the source code level of a program, we consider three definitions of behaviour:

Data-flow The values held by program variables (and then those values' attributes), and how those values move around the program.

Control-flow The paths taken through source code by runs of a program.

Timing The time taken by specific operations.

We further highlight that the case study in Chapter 7 shows that it is also necessary to consider the relationship between the three definitions.

Behaviour as Data-Flow

For systems that process a lot of data, for example those that communicate with databases or rely a lot on user input, understanding of how this data flows can be critical. For example, if the type of the data returned from a database query is wrong, a whole algorithm can fail.

In web services, analysis of data-flow is especially important because there is usually data received from some client (either a user or another program) and data that will either be inserted into or read from some persistent storage, such as a database. Because of this, checking of constraints over attributes of the data can help to detect where data-flow deviated from what was expected. For example, one might require that a function call never changes a certain value held in memory, or that the type of the data returned from a database query varies in some way depending on a parameter of the query.

In the case of the CMS Conditions Uploader, multiple computations rely on information taken from external sources, such as user input or database queries. Further, operational experience has shown that sudden changes in the way in which data is supplied by external sources can result in downtime for the service.

Behaviour as Control-Flow

Control-flow can often be considered in conjunction with timing and data-flow, since the underlying control-flow of a program can influence the timing and values held by variables directly. Conversely, data-flow can affect the control-flow for the simple reason that the conditions determining control-flow usually depend on the data present in the program. It is rare for timing to affect control-flow, since this would require some measurement of it and such measurement is normally performed only for analysis purposes, as in this thesis.

From the perspective of trying to determine which characteristics of the control-flow of a program are problematic, the control-flow is of particular interest when measured across multiple runs. Especially when considered in conjunction with data-flow and timing information, observed differences in control-flow (such as branches taken or loop iterations completed) can directly indicate problematic parts of code.

In the context of the CMS Conditions Uploader, there are cases where different paths are taken depending on the results of queries. Here, we are principally interested in the timing information generated as paths vary across multiple runs.

Behaviour as Timing

Analysing the timing behaviour of a computational system usually involves measuring the time taken by some well-defined actions. The set of possible meanings of *action* here is large and depends on 1) what our computational system is and 2) in which domain of application it operates. If our setting is a program that runs without any interaction with the physical world, like it will be in this thesis, then an action could be something at a low level like an assignment or a function call. In contrast, if we were to consider a system that communicates with the physical world in some way (ie, a cyber physical system), an action could be a composition of low-level actions that happen during a program run (communication with hardware), and actions performed by the hardware itself (such as readings taken from the environment by sensors).

If we fix the actions we care about to be function calls in a program, then the performance we want to understand might be the duration of such function calls with respect to the values held by program variables at the time of the call, hence we have a link between performance and data-flow. Alternatively, if the function calls involve some operations over a network, then we might care about the status of the environment in which the program runs during these function calls. We might even care about both the program variables and the environment. For example, if an analysis of time taken by the function calls with respect to program variable values suggests that the two are independent, it would be natural to then look at the environment.

In the case of the Conditions Uploader, timings are affected by both the surrounding network environment, and the data given by a user. Hence, we must concern ourselves with timing behaviour with respect to the data given by users, but we must also be aware of the fluctuations that can be induced by the instability of the surrounding network.

Conversely, performance could reasonably be considered as the state in a program with respect to the duration of an action. For example, given a certain accuracy of a result, the time

taken by the process that computed it is of interest. This direction will not be explored in this thesis because we do not consider systems that perform these types of computations.

Finally, given such a diverse set of possible definitions, we would like a way to analyse the performance of a program that maintains generality, rather than forcing us to use a subset of those that we have discussed.

1.3 A Typical Approach for Behaviour Analysis

If we consider a program that contains a function whose duration we want to measure and analyse with respect to the values of variables present during the program's runtime, then our idea of behaviour analysis is reduced to checking a constraint on the function's duration. For example, an engineer might profile the function so that, every time it is called, the duration is added to a timeseries along with the variable values that the engineer cares about. The resulting timeseries can then be plotted for an engineer to inspect, or a simple algorithm can inspect the raw timeseries data. Either way, the satisfaction of a constraint is checked. In the data-intensive domain, there are tools designed to do this, one of which being the CMSDBServiceStats [CMS16] framework developed at the CMS Experiment at CERN by this thesis' author.

While this is a powerful technique, and works well in the industrial setting, it has shortcomings:

- One can only perform analysis over one quantity. For example, if an engineer wants to check a performance requirement that is a boolean combination of constraints over multiple quantities (for example, values in memory and durations), this technique becomes difficult to use because the instrumentation process becomes difficult and error-prone.
- Explanation of measurements that differ from expectation is difficult, since quantities are often recorded in isolation with no relation to each other. For example, a standard profiler may record the time taken by a function each time it is called, but the information about the call site, and therefore which program paths and call trees are affected, may be lost.
- Overhead generated by system-wide profiling is often unacceptably high if software engineers do not accept the inaccuracy that comes with statistical profilers [sta]. While a comparison between the overhead induced by specification-driven instrumentation and system-wide profiling is not fair, we highlight that the lower overhead induced by specification-driven instrumentation gives opportunities for other, more in-depth analyses.

It is clear that, regarding the matters of ease of use and efficiency of analysis, we need a more sophisticated solution. Once we have such a solution, the next problem we must address is that of determining a cause in the case that a constraint is violated.

1.4 Explaining Behaviour Variations

While existing profiling techniques allow engineers to perform manual and simple analysis which could eventually lead to some notion of explanation of behaviour deviations, there is a need for a technique that automates this process for complex constraints.

We begin by briefly discussing what might constitute an *explanation* of program performance in the case that only a simple constraint is being checked. From there, we extend to a more complex case and use this as a motivation for our approach to performance analysis of programs.

1.4.1 Simple Constraints

Consider a scenario in which a software engineer wants to know the value of some variable immediately before a function call, along with the time taken by the function call. If classes of the state recorded can be identified, then inspection of any timeseries can be modified so that only the points corresponding to each class can be seen. For constraints over data-flow, manual inspection can be trickier because some types of data, such as non-primitive types, can be difficult to visualise.

Further, it is natural to ask about the control-flow leading to the point during runtime at which a measurement was taken to decide whether a timing constraint held. An engineer might ask which paths through control-flow commonly resulted in a timing or data-flow constraint being violated, and what was the difference between those and the trajectories that didn't lead to violating measurements.

In general, differences between either measured quantities or control-flow can lead to explanations; they can suggest to engineers which parts of their system, or which data given to their system, tend to cause problems.

1.4.2 Dealing with Perturbations of Timing Information

The timing behaviour of real world systems is inherently affected by perturbations due to other real world resources such as database servers, networks or even external sensors. Any measurement and subsequent analysis of constraints over such quantities must take this into account.

For example, any factors affecting the performance of a network over which a program communicates may not be easy to discover by looking solely at the source code, hence there could be slight variations in the time taken by certain actions which are not detectable via inspection of the source code.

Further, assuming that we have acknowledged such factors in the system that we are analysing, it may be the case that the timing information that we record is perturbed, sometimes slightly outside an acceptable range that we have defined. In this case, if one could consider a violation as an *almost* satisfaction (where some small perturbation occurred, but to an extent that was expected), this may allow the focus to shift to more severe violations.

Of course one must be careful about the cause of these perturbations. For example, if a function call results in calls to many other functions, many of which being measured in some way, then the process of measurement itself could cause perturbations. Hence, we acknowledge that this is a problem of which one must be aware, but in this thesis no cases are considered where instrumentation is so heavy as to cause this behaviour.

In particular, in Chapter 7, we highlight how it is possible for our instrumentation approach to generate large overhead (and therefore affect the timing that is being measured), but our

case study does not require application of our approach in this way and so we do not encounter any problems. Further, we make it clear that the type of system (web services) to which we apply our approach does not often generate the situation in which our approach results in high overhead.

1.4.3 More Complex Constraints

If the behaviour requirements are more complex, for example with constraints over multiple quantities, then explanation becomes much less straightforward. If we consider again the constraint *if* P *then* Q and want to compare either state or trajectories between runs of a program that violate this constraint and runs that satisfy it, then we run into difficulty. All runs that violate the constraint *must* satisfy P, and violate Q. But all runs that satisfy the constraint can either violate P or satisfy P *and* Q. Hence, any comparison is less straightforward because satisfaction can happen in different ways.

1.5 Turning to Formal Specification

Whenever any analyses of measurements of timing or state are performed, either by an engineer or by an algorithm, the process being performed can be reduced to checking a property. If the engineer performs the inspection, the specification that expresses the property will be in natural language and, if an algorithm performs it, the specification will probably be the condition in an if-statement.

If we consider only a single quantity and define a simple threshold over it, then we have a single constraint that can easily be checked automatically, without effort from an engineer. However, if we wish to express more complex constraints that are boolean combinations of other constraints over multiple quantities, then we need to consider a more sophisticated method for checking constraints.

We take inspiration from Runtime Verification (RV) [BFFR18], in which one checks the agreement of a run of a program with some specification, either while the program is still running or after the fact. Since RV's inception around two decades ago, work has focused on developing specification languages and monitoring algorithms, often while treating programs as *black boxes* whose runs generate *traces*, ie, abstractions of program runs. RV is seen as a *lightweight* verification technique, in comparison to Model Checking [CHV18] which has enjoyed success in certain domains but is not seen as a tool that can easily be attached to the software development process by any engineer.

A taxonomy of work in RV is described in [FKRT18]; many of the approaches to performing the lightweight verification proposed by RV that are described there involve the following:

1. Define a specification language, usually a temporal logic, in which one expresses a *property* that the computational system should hold.

2. Abstract a run of the computational system into a *trace*, a structure that contains enough information to check satisfaction of the property written in the specification language.

3. Define an algorithm that can check whether a trace satisfies a property.

The outcome of this process is usually a *verdict* that describes whether the property is satisfied by the trace. The nature of the verdict depends on the trace, the system generating the trace (does it run forever or does it stop?) and the definition of satisfaction of a property by a trace (a semantics).

1.5.1 A Low-Level Specification Language

To express complex constraints over low-level artefacts of runtime, such as function calls and variable values, we need a specification language. There is much existing work on specification languages for Runtime Verification [BLS11, Koy90, BGHS04, DSS+05], but we found that these languages tended to be at a high level of abstraction with respect to the system being checked, an undesirable property from the perspective of an engineer.

To deal with this, we designed a specification language that is easy to use to express constraints over state and time. This forms the first major contribution of this thesis: Control-Flow Temporal Logic (CFTL), a specification language designed for engineers to easily express the performance requirements of their programs, along with an associated instrumentation technique and monitoring algorithm. All of this is described in the publication [DR19b], which forms the basis of Chapter 3 and is summarised here:

A Static Model of Programs. We introduce the notion of a *symbolic control-flow graph* in Section 3.2, a statically-computable structure that encodes state changing and reachability information found in a program's source code. We use this structure in the semantics of our specification language, in our instrumentation algorithm and later in our explanation approach.

Control-Flow Temporal Logic. The specification language, Control-Flow Temporal Logic, is a linear-time temporal logic. We describe its syntax in Section 3.4 and its semantics in Section 3.5. The semantics is defined over traces with timing, variable value and control-flow information that we call *dynamic runs* (see Section 3.3). These dynamic runs consist of *states*, which are instantaneous checkpoints in a program's runtime, and *transitions*, which are the computation required to move between states. Its distinguishing characteristic is a low-level of abstraction, meaning properties have immediate meaning with respect to programs. This separates CFTL from most other specification languages. In particular, for a given specification, CFTL has no need for the mapping from events occurring at runtime to propositions in the specification that is common among other languages.

Monitoring. We consider the monitoring problem initially in the frame of *most-general dynamic runs* (intuitively, dynamic runs that contain all information that could be recorded at runtime). In Section 4.4, based on these most-general dynamic runs, we introduce a monitoring algorithm, but see that it does not scale well for two reasons:

- The dynamic run that it assumes often contains far more information than is needed to check whether a given CFTL specification holds.

- For each piece of information in the dynamic run that *is needed*, the process of determining which part of the state maintained by the monitoring algorithm should be modified to reflect the new observation is inefficient. This is because one must search through a lot of irrelevant information to find that which is needed.

We use instrumentation as a way to address these two inefficiencies.

Instrumentation. Any mechanism that we use to check the agreement of a dynamic run with a property requires that we first obtain the dynamic run. The initial monitoring algorithm presented in Section 4.4 assumes that instrumentation is not conservative and generates a dynamic run containing all information seen at runtime. This leads to a monitoring approach that does not scale well for the reasons discussed above.

In Section 4.7, we propose a far more conservative approach to instrumentation which results in a dynamic run containing less information, but still enough to check the property that has been specified. The instrumentation process we developed uses the structure of the specification being checked to determine a conservative set of program points from which to take data. When the program runs, the data taken from these points forms a filtered dynamic run that minimises the amount of work the monitoring mechanism has to do to decide whether data is really needed. Our instrumentation approach takes advantage of the fact that dynamic runs can be associated with paths through symbolic control-flow graphs.

Finally, instrumentation addresses the *intrusion problem*. This is the problem that, when performing any measurement over a computational system, we perturb the system's behaviour in some way. This is unavoidable, but perturbation to the point where violations are generated that would not be otherwise is to be avoided at all costs. Therefore, we must find a way to measure the quantities we need to inspect during the runtime of the program without too much interference. A good instrumentation approach is such a way.

Monitoring with Instrumentation. In Section 4.8, we introduce a new method for using information from instrumentation to remove the look-up overhead, resulting in an efficient monitoring algorithm for a specification language that uses (multiple) universal quantification, which is a feature of specification languages that is historically problematic for monitoring algorithms (an example being monitoring using *quantified event automata* [HRTZ18]).

1.6 Explaining Violations

Taking advantage of the low level of abstraction that CFTL admits, our explanation approach focuses on analysing program paths taken in the cases where specifications were satisfied or violated. In particular, we use established path profiling methods along with a new comparison method to construct useful explanations. Ultimately, this work is some of the first in this direction in Runtime Verification and is described in [DR19a]. This paper is the basis of Chapter 5.

Partial Semantics for CFTL. In Chapter 5, we observe that the existing semantics for CFTL reasons only about dynamic runs that represent entire program runs. In order to talk about which part of a dynamic run prevents satisfaction of a CFTL specification, we introduce the idea of a *partial semantics* which reasons over *partial dynamic runs*. Such semantics allows us to isolate a unique point in a dynamic run after which satisfaction becomes impossible. We can do this because CFTL only allows universal quantification, meaning that finite traces can violate specifications.

Quantitative Semantics. As mentioned before, when dealing with timing constraints as CFTL does, any explanation mechanism must be able to deal with slight perturbations. For this reason we introduce *verdict severity*, an extension of CFTL's semantics that captures how close to being satisfied or violated a given constraint was given some part of a dynamic run. However, rather than introducing this formally, we introduce it in the experimental setting in Chapter 7.

Path Reconstruction. To move towards a program path-based explanation technique, we extend the instrumentation method for CFTL to capture information about which path a dynamic run took through a symbolic control-flow graph. This is *path profiling*, an established approach for dynamic program analysis [BL96], but our use of it for explanation in the Runtime Verification context is new.

Path Comparison. Once we can reconstruct paths through a symbolic control-flow graph, we want to compare the paths to determine whether there are differences that can be seen to be *likely* to cause violations of CFTL properties. We introduce an approach that consists of deriving a context-free grammar from a symbolic control-flow graph, expressing a path as a parse tree with respect to this grammar, and finally comparing parse trees. We derive the context-free grammar in such a way that comparison of the resulting parse trees yields useful results about the differences between paths, including points of divergence due to branching and differences in numbers of loop iterations.

1.7 The VyPR Framework

A defining characteristic of this thesis is that the theoretical development and subsequent implementation all took place in the industrial environment in which they will be used. Hence, we will describe the VyPR framework, which checks Python programs at runtime for agreement with CFTL specifications, and VyPR2, its extension to web services. These two frameworks are described in the publication [DRF+19], which is the basis of Chapter 6 and includes a description of VyPR and its extension to web services. Further, the majority of the VyPR ecosystem along with its theoretical foundations and some example applications were presented in an extended tutorial at RV 2020 [DHJ+20]. We build on the applications described in that paper in Chapter 7.

The first implementation - VyPR. All of the theory described in this thesis is implemented in the VyPR framework. The VyPR project's website is `http://pyvypr.github.io/home/` and the source code for its components can be found at `http://github.com/pyvypr/`.

VyPR provides a library called PyCFTL with which engineers can express CFTL properties. Based on a PyCFTL specification, VyPR performs automatic instrumentation of the program under scrutiny, and then runs the instrumented program under an efficient monitoring algorithm. In the original implementation for normal Python programs, a verdict report is simply printed at the end of execution.

Extension to Web Services - VyPR2. VyPR's architecture does not fit the web service setting perfectly, so some modifications are needed. In particular, certain assumptions made in the simpler setting in which VyPR worked no longer hold in the web service setting:

- VyPR prints a verdict report *post-mortem*, but web services must be assumed to run forever so do not have a post-mortem. Further, VyPR was not designed for analysis of programs' performance over multiple runs; our work at CERN required this, so data describing satisfaction or violation of CFTL properties is made persistent by VyPR2.
- The instrumentation performed by VyPR takes place in the same process as the monitoring is performed, so memory is shared. In the web service setting, in particular in the production setting, instrumentation necessarily takes place in a separate process so memory cannot be assumed to be shared. This has consequences for instrumentation that are described in detail in Chapter 6.

Even though the initial implementation was known as VyPR2, we commonly refer to the version for web services as simply VyPR and do not often make reference to the version for local programs.

1.8 The VyPR Ecosystem

The theoretical foundations introduced in this thesis were used to develop the various tools of the VyPR ecosystem, illustrated by Figure 1.1. The core of this ecosystem is the specification, instrumentation and monitoring machinery provided by the central distribution of VyPR, found at `http://github.com/pyvypr/VyPR`. However, significant development effort went into building supporting tools, including the web-based analysis tool (described in Section 6.8) and the Python library for writing analysis scripts (described in Section 6.6).

The development of these tools marks a significant contribution to the *verdict analysis* area of Runtime Verification. In particular, we have made progress on a part of RV whose importance has been highlighted by the RV community [Reg17].

1.9 Applications of VyPR

Taking advantage of the fact that VyPR was developed in an industrial setting, we performed significant applications of it to infrastructure at CERN, the first of which being a critical

Figure 1.1: The VyPR ecosystem.

service used at the CMS Experiment. Preliminary applications have also been performed on other projects within and outside CMS, and continued use is planned within CMS after the end of the PhD work described in this thesis.

The preliminary application within CMS was, to the best of our knowledge, the first application of Runtime Verification in High Energy Physics and is described in [DRF+19]. The engineering efforts to make VyPR's explanation machinery as useful as possible and to make VyPR approachable to engineers are described in the publication [DHR+19]. This paper, together with [DRF+19], forms the basis of Chapter 7.

We will describe the application of VyPR to the CMS Conditions Uploader service [DHR+19], infrastructure used for the upload [Daw17] of Alignment and Calibrations constants (often called Conditions) ready for the processing of data originating from recorded or simulated proton-proton collisions. Given the importance of this service, a strong understanding of its performance is vital. We describe analysis of the upload service by replaying various subsets of a dataset of 35,000 uploads recorded during 2 years of runs of CERN's Large Hadron Collider.

Finally, work performed in collaboration with the Universitá della svizzera italiana led to the tool-chain PerfCI, which enables VyPR to be easily added to Continuous Integration processes for Python projects [JDH+20]. However, this work is out of the scope of this thesis.

1.10 Overview

In this chapter, we have laid out the material that will be presented by the main body of this thesis which contributes to both the Runtime Verification and Software Engineering communities, as well as the High-Energy Physics community. We have also introduced tangential work being performed at the CMS Experiment at CERN to introduce performance analysis by Runtime Verification into Continuous Integration processes. The next chapter will give an overview of the existing literature from the communities into which this thesis' contributions fit.

Chapter 2

Background

The following sections will introduce the foundations that will be used throughout this thesis alongside the related contributions from the Runtime Verification and Software Engineering communities. We will discuss the existing theory and subsequent implementations for specification of the desired behaviour of a program; monitoring of such a program to check agreement with a given specification; and determination of possible root causes given failure of a set of program runs to satisfy a specification.

While this chapter contains our survey of related work, we make a formal comparison as part of the thesis conclusion, in Chapter 8. Performing the comparison as part of our concluding remarks allows a more precise discussion of the relationship between our contributions and the existing ones, without the need to vaguely refer to ideas that will be discussed in future chapters.

2.1 The Structure of the Survey of Existing Work

We will begin by describing some existing work on specification formalisms. We will conclude our discussion of specification formalisms by considering the ease with which the existing languages could be integrated into the software development process. In doing this, we will motivate the development of our new specification language, Control-Flow Temporal Logic (CFTL), as a software development tool, rather than a language that attempts to express yet more interesting properties than existing languages.

After this, we will consider the instrumentation and monitoring problems for temporal logics in the Runtime Verification context. Much of the existing literature tends to consider these problems separately (with the exception of that discussed in Section 2.9), but one of the main contributions of this thesis is a monitoring algorithm that uses instrumentation information to improve efficiency.

Finally, we will look at the relatively underdeveloped area of Runtime Verification that involves *explaining* why a specification may not have been satisfied. The limited number of contributions from the RV community will be discussed, alongside the much more numerous contributions from the Model-Checking and Fault Localisation communities.

2.2 Behaviour Specification with Temporal Logics

To first order, the existing landscape of specification languages for Runtime Verification can be divided into the temporal logics, the rule-based languages and the stream-based languages. Due to the amount of work that has been done on temporal logics (both their application to model-checking and runtime verification), they form the richest category.

2.2.1 Preamble

Each temporal logic, like any logic, has three defining parts:

- A syntax, which is the set of rules with which one writes formulas in that logic.
- A family of *interpretations* from which to take its *models*. Such a family is a set (usually infinite) of sequences (infinite or finite, depending on the logic) of structures holding information that may change over time.
- A semantics, which is a set of rules, often encoded in a single relation, with which one decides whether or not a given interpretation *satisfies* a formula (ie, is a *model* of a formula).

The first distinction to make between temporal logics is the difference between linear-time, branching-time and *hyper*. Such distinction is made by considering the structures that form interpretations, and the abilities that individual logics have to reason over these structures.

For a linear-time temporal logic, the family of interpretations is a set of sequences, each of the form $\tau_1, \tau_2, \ldots$ such that τ_i holds information that may change with i. These logics can reason about one possible future/past. For a branching-time temporal logic, the family of interpretations is a set of *sets of* sequences, each again of the form $\tau_1, \tau_2, \ldots$ such that each τ_i holds information that may change with i. These logics can reason over multiple possible futures/pasts. For a *hyper* logic, the family of interpretations is the same as that for branching-time temporal logics, but the way in which the interpretations are constrained by the formulas is much different. In particular, rather than reasoning about multiple possible futures, formulas in hyper logics can relate what is present in multiple sequences by using *trace variables*. Following from this, one could begin a formula with $\forall\pi, \forall\pi'$ where π and π' are traces in some predefined set. This notion of *trace variables* allows one to express properties such as *observational determinism*, ie, given the same input, the result of some processes must be distinguishable only by observers with certain permissions.

We focus our attention onto linear-time logics, since this is the category into which CFTL fits. For a linear-time temporal logic, given that interpretations are sequences of structures, it is usual for such a logic to have *modal operators* whose definitions in the semantics take advantage of the index given to each structure in the sequence. For example, the popular LTL [Pnu77] has the modal operators $\mathcal{G}$ (to require that, by fixing a position i in the sequence, some demand should be met by every structure at index $j > i$ in the sequence) and $\mathcal{E}$ (to require that, by once again fixing a position i in the sequence, there is some $j > i$ whose structure meets some demand made). From this description of a linear-time temporal logic one can see

that, despite the lack of representation of *physical time* (ie, real-numbered timestamps), logical time is used. For example, with respect to some ordering on events, one can say that some propositional atom p holds immediately before some other q if the LTL formula $p \wedge \mathcal{X}q$ ("p and next q") holds.

Linear-time temporal logics that are *timed* can add to this expressiveness. There is a lack of concrete criteria that a logic must fulfil to be considered *timed*, but here we give a sensible criteria inspired by [BLS11]. A linear-time temporal logic is said to be *timed* when:

- For an interpretation $\tau_1, \tau_2, \ldots$, the τ_i are pairs $\langle t_i, \sigma_i \rangle$ for timestamps t_i in $\mathbb{R}_{\geq}$ (continuous time) or $\mathbb{Z}_{\geq}$ (discrete time) and structures σ_i.
- Its modal operators are made more expressive by the additional timing information in its interpretations.

For example, timed temporal logics can have modal operators with *bounds* so not only the index in the sequence is used in the semantics, but also the timing information held at individual positions in the sequence. In this case, given the presence of timing information in the interpretations, we say that the logic is using *physical time*.

2.2.2 Pnueli's Linear-Time Temporal Logic

With a brief characterisation of the temporal logics now given, we can introduce the existing logics themselves. We will give a brief summary of the linear-time, branching-time and hyper temporal logics. For each that we consider here, we will see their syntax, semantics and an example. We also consider the necessary modifications to their semantics for them to be used in the RV context. This often constitutes a shift from infinite word semantics to finite prefix semantics.

One of the most famous examples is Pneuli's LTL [Pnu77] which forms the basis for most other temporal logics. A subset of its full syntax is

$$\varphi := \Box\, \varphi \mid \diamond\, \varphi \mid \circ\, \varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid p \mid \text{true}$$

where $\Box, \diamond$ and $\circ$ are the modal operators and p is an *atomic proposition*.

Let Σ be a finite alphabet of *atomic propositions* and $\mathcal{P}(\Sigma)^\omega$ be the set of *infinite words* (or *ω-words*) ω whose symbols are subsets of Σ, and for which $\omega(i)$ denotes the set in $\mathcal{P}(\Sigma)$ at position i in ω (starting from 0). For example, for $\Sigma = \{p, q, r\}$, a prefix of some ω-word in $\mathcal{P}(\Sigma)^\omega$ could be $\{p, q\}\{r\}\{p\}\{p, r\}\ldots$.

The interpretations of an LTL formula φ with a fixed alphabet Σ are ω-words. The LTL semantics are described by the $\models$ relation, defined recursively in (2.1) for an ω-word ω at

position i and LTL formula φ. By this semantics, we write $\omega \models \varphi \iff \omega, 0 \models \varphi$.

$$
\begin{array}{lcll}
\omega, i & \models & \text{true} & \\
\omega, i & \models & p & \text{iff} \quad p \in \omega(i) \\
\omega, i & \models & \varphi_1 \vee \varphi_2 & \text{iff} \quad \omega, i \models \varphi_1 \text{ or } \omega, i \models \varphi_2 \\
\omega, i & \models & \neg\varphi & \text{iff} \quad \omega, i \not\models \varphi \\
\omega, i & \models & \Box\, \varphi & \text{iff} \quad \text{for every } j \geq i : \omega, j \models \varphi \\
\omega, i & \models & \diamond\, \varphi & \text{iff} \quad \text{there is some } j \geq i : \omega, j \models \varphi \\
\omega, i & \models & \circ\, \varphi & \text{iff} \quad \omega, i+1 \models \varphi \\
\omega, i & \models & \varphi_1\, \mathcal{U}\, \varphi_2 & \text{iff} \quad \text{there is some } j \geq i : (\omega, j \models \varphi_2 \\
 & & & \qquad \text{and for every } i \leq k < j : \omega, k \models \varphi_1)
\end{array}
\tag{2.1}
$$

Based on this semantics, as an example we take the prefix $\omega = \{p, q\}\{r\}\{p\}\{p, r\}\ldots$ and consider the formula $\varphi \equiv \Box\, p$ that expresses a safety property (hence can be violated by a finite prefix of an ω-word, but cannot be satisfied by a finite prefix). This is violated because we can set $i = 1$ and ask whether $\omega, 1 \models \Box\, p$. For this to be the case, we must have $p \in \{r\}$ which is clearly not the case, so we have $\omega \not\models \Box\, p$.

We consider instead a simple liveness property, $\varphi \equiv \diamond\, r$. Liveness properties can be satisfied by finite prefixes of ω-words, but cannot be violated by finite prefixes because there is always a chance of a satisfying extension. Hence, the prefix $\omega = \{p, q\}\{q\}\{p\}\{p, r\}\ldots$ gives $\omega \models \diamond\, r$ because we can check $\omega, 0 \models \diamond\, r$ and take $j = 3$ such that $\omega, 3 \models \varphi$ because $r \in \{p, r\}$.

In order to use LTL as a specification language for the RV context, it is necessary to extend its truth domain. In particular, in RV the word whose satisfaction of an LTL formula we check is a representation of a run of a program, usually called a *trace*. Hence, if we check such a trace as the program is running, we must assume that we have seen only a finite prefix. With the LTL semantics in (2.1), the definitions of modal operators assume that a future always exists; in online RV this is not the case.

One approach, from Kamp [Kam68], is to simply give $\bot$ if the future to which a modal operator refers does not exist (interpreting the conventional modal operators as *strong*, and introducing new *weak* operators to default to $\top$). For example, $\varphi \equiv \circ\, p$ gives $\bot$ over the finite word $\{p\}$ because, to check $\omega, 0 \models \varphi$, the position $0 + 1$ does not exist. The utility of this approach depends on the domain of application.

Further, Manna and Pneuli [MP95] discuss the introduction of strong and weak versions of modal operators to yield a 4-valued truth domain for FLTL_4. The truth domain here is $\{\top, \bot, \top^p, \bot^p\}$, where $\bot^p$ and $\top^p$ are yielded by checking a finite prefix with a formula with a strong or weak version of a modal operator respectively.

Bauer et al. [BLS10] conclude that it is necessary to extend to a three-valued truth domain. RV approaches in general consider truth domains of at least 3 values. Where the standard semantics in (2.1) used the two-valued domain $\mathbb{B} = \{\top, \bot\}$, the three-valued domain used by Bauer et al. is $\mathbb{B}_3 = \{\top, \bot, ?\}$ where ? is intuitively *inconclusive*. The second conclusion of Bauer et al. is to move from a semantics relation to the semantics function in (2.2), in which

$u\sigma$ denotes the concatenation of the finite prefix $u \in \Sigma^*$ with the ω-word extension $\sigma \in \Sigma^\omega$.

$$[u \models \varphi] = \begin{cases} \top & \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases} \tag{2.2}$$

Their intuition is that, if it cannot be concluded that every extension of a finite word to an ω-word will yield a verdict in the $\mathbb{B}$ domain, then we conclude the new third value ?.

Before we investigate the timed variants of LTL, we remark that there are numerous untimed extensions of LTL. Such extensions could include enrichment of the atomic propositions that can be used (for example, to include equality checks) and the addition of quantification to variables given to predicates (First-order LTL [KBC16]). Another extension, CaRet [AEM04], allows LTL to be used to describe constraints over function *calls* and *returns*.

2.2.3 Timed Variants of Linear Temporal Logic

We will examine two popular extensions of Pnueli's LTL that introduce an ability to reason about time. These variants are Timed Linear-time Temporal Logic (TLTL) [BLS11] and Metric Temporal Logic (MTL) [Koy90], and each introduces time into its modal operators in a slightly different way. In particular, TLTL introduces operators which ultimately allow one to constrain the time elapsed between points and MTL introduces discrete-time bounded versions of the standard modal operators used by LTL.

Fixing an alphabet Σ, TLTL is defined over *timed* ω-words from $(\Sigma \times \mathbb{R}_{\geq})^\omega$, hence infinite sequences of pairs $\langle a, t_i \rangle$. This is in slight contrast to the words over which LTL is defined since TLTL considers individual *events*, rather than sets of *atomic propositions*. The syntax of TLTL formulas is given in 2.3.

$$\varphi = a \mid \triangleleft_a \in I \mid \triangleright_a \in I \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \mathcal{X}\, \varphi \tag{2.3}$$

The semantics, given in (2.4) for a TLTL formula φ and a timed ω-word ω, require *event recording variables* x_a which give the timestamp at which the event a was last observed ($\bot$ if it wasn't); *event predicting variables* y_a which give the timestamp in the future at which the event a will next be observed; and *clock valuation functions* γ_i which give the values of x_a and y_a at the position i in a timed ω-word.

$$\begin{array}{lcll} \omega, i & \models & \text{true} & \\ \omega, i & \models & a & \text{iff} \quad a = \omega(i) \\ \omega, i & \models & \varphi_1 \vee \varphi_2 & \text{iff} \quad \omega, i \models \varphi_1 \text{ or } \omega, i \models \varphi_2 \\ \omega, i & \models & \neg\varphi & \text{iff} \quad \omega, i \not\models \varphi \\ \omega, i & \models & \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} \quad \text{there is some } j \geq i : (\omega, j \models \varphi_2 \\ & & & \qquad \text{and for every } i \leq k < j : \omega, k \models \varphi_1) \\ \omega, i & \models & \triangleleft_a \in I & \text{iff} \quad \gamma_i(x_a) \in I \\ \omega, i & \models & \triangleright_a \in I & \text{iff} \quad \gamma_i(y_a) \in I \end{array} \tag{2.4}$$

The extension of TLTL to a 3-valued truth domain is done in the same way as for LTL, that is, by considering extensions of finite timed words to timed ω-words.

Finally, Metric Temporal Logic allows one to reason about time by adding bounds to the standard modal operators from LTL. The syntax is given by

$$\varphi := p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \,\mathcal{U}_I\, \varphi \mid \varphi \,\mathcal{S}_I\, \varphi$$

with the semantics given in (2.5). The semantics for MTL consider time as being a continuous underlying structure, hence timed words are defined slightly differently as maps $\omega : T \to \Sigma$ for $T \subseteq \mathbb{R}_{\geq}$, ie, from points in continuous time to *events* in the alphabet Σ.

$$\begin{array}{lcll} \omega, t & \models & p & \text{iff } p = \omega(t) \\ \omega, t & \models & \varphi_1 \vee \varphi_2 & \text{iff } \omega, t \models \varphi_1 \text{ or } \omega, t \models \varphi_2 \\ \omega, t & \models & \neg\varphi & \text{iff } \omega, t \not\models \varphi \\ \omega, t & \models & \varphi_1 \,\mathcal{U}_I\, \varphi_2 & \text{iff there is some } t' \geq t : \\ & & & \quad (\omega, t' \models \varphi_2 \text{ with } t' - t \in I \\ & & & \quad \text{and for every } t \leq t'' < t' : \omega, t'' \models \varphi_1) \\ \omega, t & \models & \varphi_1 \,\mathcal{S}_I\, \varphi_2 & \text{iff } \ldots \end{array} \tag{2.5}$$

Alternative methods that have been proposed include *freeze quantification* [AH94], which is an additional operator for logics over timed words that takes the current time and stores it in a variable; and clock variables [AH93], which are assumed to be given the value of the current time (continuous).

We conclude our discussion of linear-time temporal logics with an illustration of the difference between TLTL and MTL. Suppose that we would like to express the property that, *globally, whenever p holds, q follows in at most* 5 *units of time.* In TLTL, one might write

$$\Box(p \implies \triangleright_q \in [0, 5])$$

then in MTL one might write

$$\Box(p \implies \mathit{true}\, \mathcal{U}_{[0,5]}\, q).$$

Here one can see the difference in how physical time is accessed in formulas.

2.2.4 Logics for Concurrent Systems

The verification of concurrent systems, as opposed to serial systems, presented problems which gave rise to the development of the Computation Tree Logic (CTL) [CE82], introduced by Clark and Emerson to overcome the inability of existing logics (such as LTL) to describe concurrency. Concurrency is modelled by first describing a system as a Kripke structure [Kri63] $\mathcal{K}$ (a labelled transition system augmented by a function λ from states to sets of propositions that are true in those states) and then considering a *computation tree* $\mathcal{T}$. $\mathcal{T}$ is an *unfolding* of $\mathcal{K}$, that is, every path from the root of the tree, which is the starting vertex of $\mathcal{K}$, to a leaf can be identified with a path through $\mathcal{K}$.

The semantics of CTL is defined over the entire structure of $\mathcal{T}$. To see the extra expressiveness that CTL brings compared to that of a linear-time logic such as LTL, we fix an alphabet Σ. We observe that an ω-word $\omega \in \mathcal{P}(\Sigma)^\omega$ over which some LTL formula φ is evaluated can be seen as a path through an unfolding $\mathcal{T}$ of a Kripke structure. This can be seen by taking the states $s_1, s_2, \ldots$ of the path through the unfolding $\mathcal{T}$ and constructing an ω-word from $\mathcal{P}(\Sigma)^\omega$ by applying λ to each s_i. Hence, the structures over which LTL formulas reason are parts of the structures over which CTL reasons, however CTL is not strictly more expressive than LTL, since one cannot express *stability* ($\diamond \Box\, p$) in CTL.

CTL formulas have the syntax given by the grammar

$$\varphi := \text{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid E\mathcal{X}\ \varphi \mid E\mathcal{G}\varphi \mid E\mathcal{U}\varphi$$

Here, E means *on some path*. Additional operators can be defined using combinations of these basic quantifiers, such as the *for all paths* quantifier.

Notice that it is not possible to generate a modal operator such as *next*, $\mathcal{X}$, without a path quantifier. This is because, in an unfolding of a Kripke structure, there must be some quantification over on which paths from a state a constraint must apply. The semantics of the standard modal operators that are shared with LTL are the same. Hence, $\mathcal{AG}\varphi$ means "on every path, φ is always true".

Emerson and Halpern proposed an extension of CTL, called CTL*, in 1986 [EH86]. This version is strictly more expressive than CTL because of the addition to its semantics of *path formulas*, which are subformulas without a path quantifier. The well-formedness condition of this logic requires that any subformula without a path quantifier must be given a path quantifier by some parent formula, meaning subformulas without path quantifiers are implicitly given a path over which to be evaluated by the outer path quantifier.

Hyperproperties

Taking a different view of concurrent systems (or any system in which multiple *entities* work together), logics that reason over separate traces, rather than interleaved traces (which is what happens when we consider a computation tree), are seen as describing *hyperproperties*. One such example is *HyperLTL* [CFK$^+$14], an extension of LTL to use *trace variables* (variables whose values are taken from indices in some traces) and then enable quantification over traces via quantification over trace variables. Specification languages for hyperproperties can also be used to capture properties that span multiple runs of the same system.

An example hyperproperty which was given earlier is *observational determinism*; the property that, if an observer does not have privileges to see a specific input, then the traces that the observer sees must not enable inference of that input. Finally, an in-depth account of the existing work on hyperproperties (including specifications for observational determinism and a discussion of HyperLTL) can be found in [CFK$^+$14].

2.3 Rules and Streams

Moving away from temporal logics, we now briefly investigate alternative approaches to specification of program behaviour. We will first look at rule-based specification languages and then at stream-based specification languages.

Rule-based specification languages, such as EAGLE [BGHS04], allow properties to be constructed via systems of equations that define temporal operators recursively. For example, one can express the LTL formula $\Box(x > 0 \rightarrow \diamond\, y > 0)$ using the rule system

$$\begin{aligned}
&\underline{\max}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \circ\ \texttt{Always}(F)\\
&\underline{\min}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \circ\ \texttt{Eventually}(F)\\
&\underline{\text{mon}}\ M_1 = \texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))
\end{aligned}$$

where, with the logic's support of recursion, only the *next* operator is needed. Further, the max operator can be seen as giving the *maximal* solution of the recurrence relation it accompanies and the min operator the *minimal* solution. The mon operator indicates which rule should be used as the *entry-point* for the monitor (indicating implicitly that the rules with max and mon are just to be used by the monitor).

Alternatively, stream-based specification languages, one of the most notable being LOLA [DSS+05], are based on *stream equations*. One of the defining characteristics of such specification languages is that the notion of verdict can be extended to non-boolean types.

In particular, the stream equations in a LOLA specification have the form

$$\begin{aligned}
s_1 &= e_1(t_1, \ldots, t_m, s_1, \ldots, s_n)\\
&\vdots \quad\ \vdots\\
s_n &= e_n(t_1, \ldots, t_m, s_1, \ldots, s_n)
\end{aligned}$$

where the t_i are *stream variables* of type T (they are sequences of values taken from T) and s_i are stream variables whose values are determined by expressions e_i over stream variables s_i and t_i.

As an example of verdicts extending out of standard (2 or more valued) truth domains, one can count the number of times an input stream t of type $\mathbb{B}$ contains an odd number:

$$s = s[-1, 0] + (t \bmod 2)$$

where $s[-1, 0]$ means the previous position in s, defaulting to 0 if no such position exists. Hence, the property is captured by adding 1 to a count (a separate stream, which maintains a history of counts) whenever an odd number is found, which is done by taking the value of t modulo 2.

2.4 Plans

Leaning towards the automata-based specification side of RV is the work on *observers* [BBKT05]. This allows software engineers to supply *plans* (expressions that can be translated into automata), which are then translated into sets of automata called *observers*. This work forms an

early contribution to the RV community.

2.5 Automata

So far, we have considered properties expressed mostly by temporal logics, stream equations and rules. While monitoring, in particular for temporal logics, often involves a translation of the specification expressing the property into some kind of automaton that expresses the same property, some specification formalisms involve expressing properties with the direct use of automata.

2.5.1 Quantified Event Automata

The most notable example of automata-based specification is the Quantified Event Automata (QEA) [BFH+12]. This formalism takes away the process of writing specifications as formulas in some logic, instead requiring the construction of automata with some quantification as additional structure.

The QEA formalism addresses the problem of monitoring properties involving data. In particular, it considers traces τ taken from some alphabet Σ, but where each *event* τ_i contained within τ is parameterised by data from some type T. For example, a trace τ could look like

$$\mathsf{open}(f_1)\ \mathsf{close}(f_1)\ \mathsf{open}(f_2)\ \mathsf{close}(f_2)$$

where the alphabet of this trace is $\Sigma = \{\mathsf{open}(f), \mathsf{close}(f)\}$ and each event is parameterised with respect to the type $T = \{f_1, f_2\}$. For example, $\mathsf{open}(f)$ has parameter f whose value can be taken from T.

Such parameterisation gives rise to a natural quantification. For example, suppose some automaton encodes the property φ over the alphabet Σ while leaving all parameters free (ie, without a value). Then it makes sense to write $\forall f \in T : \varphi$ to require that φ holds when each value taken from T is given to the parameters in the events from Σ over which it is written. It also makes sense to write $\exists f \in T : \varphi$.

2.5.2 DATEs

The DATE (Dynamic Automata with Events and Timers) specification formalism [CPS09a] allows one to construct automata that can make transitions upon receiving certain *events*, but only if certain conditions defined on the transitions hold. Further, instructions can be assigned to transitions, so that taking the transition executes the instruction over data that is global to the automaton. For example, one could maintain a counter and increment it each time a transition is taken that leads to an certain state. The value of this counter can then later be used to decide on a transition to be taken. One can also use the values of clocks to decide on which transition to take.

A DATE is formally a set of automata that can communicate with each other via *channels*. The events that are considered by the DATE formalism can also result in the instantiation of

new automata. Hence, automata can be instantiated by one automaton sending an event to another via a channel.

This dynamic instantiation of automata is useful for monitoring multiple instances of a property. For example, a property describing the expected behaviour of a single user in a system (in which case there would be an automaton instantiate for each user).

Despite the abstract nature of the *events* that are allowed on transitions in DATEs, such events can be the relatively low-level events taking place during the execution of a computational system, such as method calls and exceptions. In terms of tool support, one example of a tool that uses the DATE formalism is LARVA [CPS09b].

2.5.3 ppDATEs

Work that combines the DATEs formalism with Dynamic Logic [dyn79] is that on ppDATEs [APS12]. This work takes some of the characteristics of Dynamic Logic (namely, the ability to express pre and post-conditions of computation) and allows them to be added to transitions in DATEs. For example, one can require a pre-condition to hold before a transition is taken, and a post-condition to hold after it is taken.

The extension of DATEs to ppDATEs was performed as part of the development of the STARVOORS framework [ACPS17], which attempts to prove pre and post-conditions of specifications in order to reduce monitoring overhead. Given a specification and a program, STARVOORS performs the following procedure:

1. It takes specifications expressed in ppDATEs and uses the KeY tool [ABB+05] to attempt to statically prove the pre and post-conditions expressed by the specifications, given the program under scrutiny.
2. If a property can be entirely proved, it is removed from instrumentation and monitoring.
3. If a property can be *partially proved*, a specialised property is generated that does not require data from part of the system at which properties have been proven.
4. If a property cannot be proved, it must be monitored entirely.
5. The new property is then monitored using LARVA [CPS09b], a tool that takes properties expressed in multiple specification formalisms, one of which being DATEs.

2.6 Complex Event Processing for RV

Additional contributions to the specification languages area of RV include *BeepBeep* [Hal16], a visual specification approach that enables software engineers to monitor streams of data by building pipelines of operations to be performed on the streams. In particular, this work combined RV with Complex Event Processing (CEP).

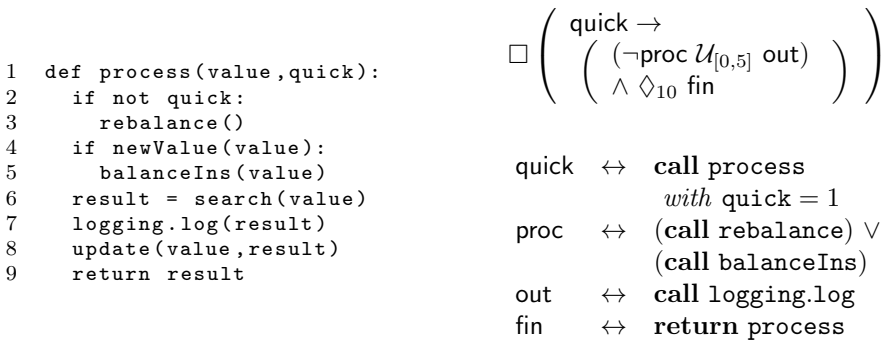


Figure 2.1: An instrumentation mapping constructed between a program and a specification.

2.7 The Specification Language as a Development Tool

A major topic considered in this thesis is that of seeing specification languages as a tool for software development. In order for a specification language to be useful as a tool, it must be straightforward to express the properties that are of interest to engineers. Reaching this point often involves sacrificing expressiveness that would allow expression of very complex properties, but also make for a convoluted language with lengthy formulas that describe relatively simple properties.

To highlight this point, we take the timed specification language Metric Temporal Logic and consider the steps required to use it in a software development project. Assuming that a software engineer has some knowledge of what they would like to check in their program, their first step would be to consider the events in their system and decide how to define an abstract alphabet such that each symbol in that alphabet would somehow map to the events in their system. The process of deciding which events in the system are relevant to the specification being checked forms part of *instrumentation*, while *instrumentation* can also involve structuring of the events observed during a run of the system in order for them to be used in monitoring [KVK+04]. Given this use of instrumentation as a way to map events occurring at runtime to the parts of a specification to which they are relevant, an *instrumentation mapping* is often discussed in the RV community and is described in more detail in Section 2.9.

Figure 2.1 illustrates the construction of such a mapping. The MTL specification at the top right is mapped to the events generated at runtime by the program on the left by the instrumentation mapping at the lower right. In this case, we assume a simple language for construction of the mapping, where for example one writes rules using a $\leftrightarrow$ whose left-hand-side is an atomic proposition from the specification and right-hand-side is an expression in terms of *events* which we loosely define now. Events are of the form **call** `func` to indicate the call of a program function called `func`, or **return** `func` to indicate a return. Further, expressions using *call* can be augmented using the *with* keyword so that only those events with a certain parameter value will be used.

2.7.1 Practicalities of an Instrumentation Mapping

The notion of an instrumentation mapping from a high-level specification to a program allows the specification language to be expressive. For scenarios concerning properties over, for example, API usage across a codebase, an instrumentation mapping is useful because it avoids the requirement for a specification to be rewritten for each function. Rather, in such cases a higher level specification language with an instrumentation mapping would be better suited. However, when low-level properties are required, an instrumentation mapping introduces two main difficulties to the development process if the specification language is to be considered a development tool.

The first such difficulty is that extra information, the instrumentation mapping, must be maintained alongside the program and the specification. The extra effort required for this activity may prevent adoption of the specification language. The second, not orthogonal to the first, is that knowledge of the instrumentation mapping is required to understand the specification. This fact gives rise to opportunities for error, for example if a specification is failing but there is a misunderstanding of the correspondence between the program, the instrumentation mapping and the specification.

2.7.2 Expressiveness versus Utility

Many existing specification languages allow expression of highly non-trivial properties through the use of complex modal operators. For example, Metric Temporal Logic allows nesting to an arbitrary depth of operators such as $\mathcal{U}_I$, the *bounded until* operator described in Section 2.2.3. While the ability to nest such operators allows the expression of non-trivial properties, the resulting formulas can be difficult to understand for engineers.

Further, the possibility for specifications becoming difficult to understand could make the task of maintaining a separate instrumentation mapping more difficult. For example, maintenance of an instrumentation mapping can require understanding of what the specification for which it was written captures. If the instrumentation mapping is not easy to understand, the effort required to maintain it is likely to be intrusive into the development process. Chances of such intrusion would again prevent adoption of the specification language.

2.7.3 Ease of Analysis

A significant contribution made by this thesis is work on how to analyse the results of monitoring a program run. Much work has been done on deciding which kinds of results the runtime verification process could generate, but there is little work on providing a translation from the formal semantics to a representation that may be more useful to software engineers.

The importance of a translation lies in the fact that the analysis facilities that are available alongside a specification language contribute just as much to the decision on whether to adopt the language as the language itself. This is clearly seen if one imagines the process through which a software engineer is likely to go when using specification as a development tool:

First, the specification is written with respect to the program being checked. The program

is then run by the software engineer (possibly in some testing infrastructure) and verdict information is obtained based on the semantics and truth domain of the specification language being used.

The problem here is that the representation of information obtained at runtime in terms of formal semantics is not always the most useful to software engineers. To see this, consider the LTL formula

$$\phi \equiv \Box\ (p \rightarrow \Diamond\ q).$$

If there is some trace τ extracted from a program run with $\tau \not\models \phi$, interpretation of this violation purely in terms of the formal semantics would require us to first look at what is needed for $\tau \not\models \Box\ \phi$. We have $\tau \not\models \Box\ \phi$ if and only if there is some position i in the trace τ such that $\tau, i \not\models \phi$. Given $\phi \equiv p \rightarrow \Diamond\ q$, we have $\tau, i \not\models p \rightarrow \Diamond\ q$ if and only if $\tau, i \models p$ and $\tau, i \not\models \Diamond\ q$. Finally, we have $\tau, i \not\models \Diamond\ q$ if and only if there is no position $j > i$ such that $\tau, j \not\models q$.

Therefore, given the relevant positions i and j, failure of φ can happen if both $\tau, i \models p$ and $\tau, j \not\models q$. Further, p and q are from an abstract alphabet and so the instrumentation mapping is required to understand what both of these conditions mean. A process like this for analysis of the monitoring results with respect to a simple formula with only two atomic propositions shows that analysis with respect to the formal semantics may not be the best approach for engineers. If understanding of the instrumentation mapping is required not only to interpret the meaning of a specification, but also for analysis of monitoring results, then a translation from the formal semantics may be useful.

2.8 Monitoring Online and Offline

Monitoring a run of a program for agreement with some specification is a problem whose solution depends strongly on the domain. Some domains, such as cyber-physical systems (flight control software, etc) [SSA+19, BBL+20, ZBK20], require monitoring at runtime in order to allow the fast correction of errors. Other domains, such as the one in which this thesis is based, require monitoring for later analysis. In these cases, some solutions even move towards recording a trace and monitoring offline [CPS09b]. There is a clear tradeoff: online monitoring allows for faster determination of whether a system satisfied a property but threatens to induce too much overhead (caused by a monitoring procedure) and unacceptably perturb the system being measured. Offline monitoring usually has a lower risk associated with inducing unacceptable overhead at runtime because the only activity to perform is that of recording the trace (rather than executing the monitoring procedure). However, the disadvantage of this approach is that measurements are only available much later. Further, depending on how the trace is recorded (whether it is written to a file, sent to a database, sent to a remote server, or some other method), the act of recording each measurement in the trace could threaten to induce unacceptable overhead.

The overhead problem has been addressed, one example being [ZELS19] in which the boundaries between where monitoring can be synchronous and asynchronous are estimated statically. This work follows the intuition that asynchronous monitoring induces less overhead since the system does not wait for a verdict. The Clara tool [BLH10] attempts to deduce parts of a

program that may not require monitoring based on sequences of increasingly complex static analyses. This will be discussed in more depth in Section 2.9.3.

In the offline monitoring context, the problem to address is space overhead. This can be a problem when traces are detailed, and when traces are derived from long-running programs. Work presented in [BCE+14] addresses this issue. The issue of online monitoring inducing sometimes unacceptable runtime overhead is also used as motivation in [CPAM09].

2.8.1 Dependence on the Specification Language

Monitoring algorithms (and underlying data structures) differ depending on the specification language. If the specification is an automaton (such as a quantified event automata), then monitoring involves stepping through the automaton while maintaining any surrounding state (for example, quantification over values).

If the specification, on the other hand, is some formula written in one of the zoology of temporal logics, then a monitoring algorithm is likely to run based on a synthesised automaton. Such an automaton would be based on the formula [VW94, GO01] to the extent that the language it accepts would be precisely the set of ω-words captured by the formula.

Given such an automaton, monitoring would essentially be the process of running events received from the monitored system through that given some definition of how the events in the monitored system would map to the alphabet (and, therefore, the symbols appearing in the formula) of the automaton. In particular, synthesis techniques often yield *Generalised Büchi Automata*, that is, finite state automata characterised by their acceptance condition: an ω-*word* results in traversal by the automaton of at least one state in a set of accepting states infinitely often [Büc90]. A post-processing step transforms the generalised Büchi automaton to a standard Büchi automaton.

When monitoring for properties expressed using timed temporal logics, such as Timed Linear Temporal Logic, one approach is to use automata with clocks [BLS11]. Alternatively, Ho et al. [HOW14] proposed an approach for monitoring Metric Temporal Logic formulas that involves transforming arbitrary MTL formulas into LTL formulas and then taking advantage of existing methods for automata-based monitoring of LTL formulas.

In Section 2.9.3, we describe some approaches for using information obtained from static analyses to optimise the monitoring process. These approaches often assume that a specification can be expressed in the form of an automaton.

2.8.2 Problems with Quantification

Monitoring a system for agreement with a specification that uses quantifiers is often problematic. When quantifiers are used, this requires evaluation of some inner, quantifier-free formula at each value captured by the quantifiers. If the trace being checked is assumed to be available immediately (in the case of Offline Runtime Verification, or monitoring of infinite words outside the context of Runtime Verification), determining the values captured by quantifiers is not as big of a problem. The problem comes in the context of Online Runtime Verification, where we may only assume that we have a *finite prefix* of whatever trace we will ultimately be checking

(whether that be infinite or finite). This leads to a requirement for incremental construction of the sets of values that quantifiers capture. This incremental construction of the sets of values that quantifiers should capture is addressed particularly in the work on QEA [BFH+12, HRTZ18], in which both universal and existential quantifiers are allowed.

2.9 Static Analysis and Instrumentation

Most work on specification languages considers the problem of checking whether a trace τ satisfies the property expressed by a specification φ. In doing this, the program under scrutiny is seen as a black box, that is, the problem of extracting τ from the program is not considered.

If the program is no longer seen as a black box (ie, its source code is available), or a model of the program is available, then static analysis can be used to optimise the monitoring process. For the purpose of discussing existing approaches to using static analysis to optimise monitoring, we refer to the problem of deciding from which parts of code to extract data at runtime as *instrumentation*. However, we will also refer to the use of data obtained *during instrumentation* to optimise monitoring. We will now discuss various work in these two directions.

2.9.1 Instrumentation Languages

The usual solution to the instrumentation problem for a given specification language and type of system involves definition of a language in which one can write a translation from the atomic propositions in a formula to the events that happen during the runtime of a program. Figure 2.1 shows an instrumentation mapping built using a simple language which is capable of capturing a limited range of behaviours that could be exhibited by a program (function calls and returns).

Despite the difficulties found in maintenance and comprehension from the engineer point of view, the conventional approach of separating instrumentation from the specification language has the clear benefit that the same specification language can be used to express properties at different levels of granularity over a system. For example, the property $\Box\ (p \rightarrow \Diamond\ q)$ can be applied to function calls, object instantiations, or behaviour not directly found in code such as network behaviour.

A tool that is notable for its use of an instrumentation language to separate the specifications from the program being monitored is JAVA-MAC [KVK+04]. The approach there is to have software engineers define their specifications in a high-level language, then use a specification language to define how the events generated by the monitored program at runtime relate to the specification. To enable this, two languages are used: the *Primitive Event Definition Language* (PEDL) and the *Meta Event Definition Language* (MEDL). This approach has the benefits described above.

2.9.2 Existing Implementations of Instrumentation

Implementations of instrumentation can be characterised by how much separation there is between the program under scrutiny and the logic that checks the specification. If there is less separation, then each additional piece of code added as an instrument may be required to

perform some complex logic related to checking of the specification. If there is more separation, then each additional piece of code will likely just pass a message to a separate mechanism. Instrumentation with less separation places significant pressure on the robustness of the code that monitors the program; if there is an error here, the program being monitored will go wrong and this is unacceptable.

We now turn our attention to the established tools for instrumentation. We will look briefly at Java and C/C++, before highlighting the lack of work for Python. We will also give justification for the lack of work on Python instrumentation, but highlight where there is nonetheless a need for it.

Java

Given the focus of research in Runtime Verification in Java, by far one of the most popular instrumentation approaches is to use AspectJ [Asp]. This extension of Java works by adding pieces of code at key points in a Java program. One of the most powerful features is the ability to add (or, in AspectJ terminology, *weave*) additional instructions at *joinpoints* (points in control-flow to which multiple paths lead). The notion of a joinpoint is a generalisation and captures events such as function calls and variable accesses. Such machinery is made possible by the Java Virtual Machine's powerful internals. Any weaved code can send messages to a monitoring mechanism; indeed this is how many Runtime Verification tools (or possibly languages and accompanying tools) [JMLR12, CPS09b, dH05b, dH05a] for Java work. Further, ParTrap [BCBC+18] is a tool for monitoring Java with respect to specifications written in a language inspired by Python. Finally, the BISM tool [SKF20] enables sophisticated, comprehensive instrumentation of Java programs.

C and C++

Instrumentation implementations for C and C++ include the RMor [Hav08] framework developed at JPL, which uses the Cil tool [cil] for C program transformation. In this case, an instrumentation mapping is defined as part of the specification and this is used alongside Cil to modify the program under scrutiny.

Python

Widely-used and stable implementations of instrumentation of Python code were, before this work, non-existent to the best of our knowledge. There are some small libraries available, but we have found none that are actively maintained. Further, Python has no widely-used version of aspect-oriented programming, but does provide powerful code analysis libraries as part of its standard library.

The standard Python interpreter (CPython) provides some powerful introspection features, such as *trace functions* [pytc], whose intended use is that of building debuggers and profilers. However, the overhead induced by this approach can be high since trace functions are executed for every statement in the program by the interpreter. The alternative approach, which is taken in the implementation described in this thesis, is to use Python's native libraries for the

construction of abstract syntax trees of programs. These trees are modified by instrumentation and then the native libraries are used to compile the modified trees to Python bytecode.

We conclude this discussion of instrumentation techniques by highlighting the biggest problem in instrumenting for Python: its type system. Python is a very simple language in which to write programs, but static analysis is famously problematic because of the weak type system, alongside which variables (including functions, since these are first-class citizens in Python) can be passed around without any guarantees that a variable will always hold the same type. With this in mind, instrumentation approaches that rely solely on static analysis will generally lead to over-approximation. This thesis does not try to improve on this situation because in our use cases imperfect instrumentation was not a problem.

2.9.3 Static Analysis and Residues

A significant contribution of this thesis is the idea of using information obtained during static analysis to optimise the monitoring process for a given specification. Performing static analysis with the aim of optimising monitoring has already attracted attention from the RV community.

The Clara tool

The first example to consider is the Clara tool [BLH10] for Java. Clara uses sequences of static analyses [BLH12] in order to reduce specifications (automata) to *residues*. The monitoring of these residues induces less overhead than monitoring for their complete counterparts, and the same violations will be caught (since any part of a specification that is no longer monitored has been proven to hold during the static analyses). The procedure employed from the supply of specifications through to monitoring at runtime is as follows:

1. Determine the set of points at which data should be recorded at runtime. Given that the setting is Java, this involves AspectJ. The points determined to be of importance for a given specification are known as *shadows* (each *joinpoint*, a point in code at which some code woven by AspectJ should execute, has a corresponding *shadow*).

2. From this initial set of points, a sequence of static analyses are performed with the goal of reducing the number of points at which to record data.

3. Each static analysis reduces the number of locations in the *weaving plan*, ie, the set of places into which code should be weaved by AspectJ for execution at runtime.

The result of this procedure is that less additional code is executed at runtime to perform monitoring because less has been woven into the program under scrutiny.

We now discuss three of the static analyses that are employed: *quick check*, *orphan-shadows analysis* and *nop-shadows analysis*. *Quick check* simply inspects the program under scrutiny to find any code (ie, a joinpoint) that could generate an event that causes an immediate violation of a property. *Orphan-shadows analysis* consists of applying *quick check* per object. Hence, if a specification says that a *connection* object should never call a certain pattern of methods and this can be shown to be impossible by static analysis, then this object does not have to be monitored. Finally, *nop-shadows analysis* is control-flow sensitive, and involves determining

whether following certain paths through control-flow can lead to an object reaching a certain *typestate* (essentially, a description of which actions can be performed on the object at a given time). This analysis can be used to see whether certain paths through control-flow can lead to objects entering erroneous typestates (ie, states resulting from incorrect use of objects' methods).

Finally, the CLARA tool has been combined with the LARVA tool (to form CLARVA) in order to take a Model-based approach to Runtime Verification [ACP20]. In particular, this work is done in the setting in which the source code of the program under scrutiny cannot be assumed to be available, so a model is used instead.

Sampling-based Monitoring

One disadvantage of identifying all of the points in a program at which to record data at runtime, as identified by Bonakdarpour et al. [BNF11], is the effect that they call *event bursting*. This effect is a result of the program points identified by the instrumentation mapping for a specification being traversed at runtime often enough to overwhelm the mechanism responsible for monitoring the program. This is especially a problem with synchronous runtime verification, in which the monitoring mechanism blocks the program under scrutiny. In this case, if the mechanism is overwhelmed, the behaviour of the program under scrutiny suffers. The solution proposed by Bonakdarpour et al. is to perform *sampling-based* monitoring using a more conservative instrumentation strategy.

This strategy aims to determine key points in a program between which no changes that can cause a property violation can occur. This amounts to determining parts of the program in which no instruments need to be placed, hence making event bursts less likely.

Overhead-aware Monitoring

A common approach to instrumentation in settings where overhead must not only be minimised but kept strictly below an upper bound is to diverge from the conventionally static analysis-based approaches, introducing *overhead-aware* monitors [ZELS19]. This work is an attempt to statically determine where monitors may induce too much overhead.

Code Transformation

When monitoring for specifications that define properties over the source-code level behaviour of programs, constructs in code that are of interest to efforts to reduce overhead are loops. For example, a given specification may require a single measurement (or far fewer measurements than there are iterations of the loop). In this case, instrumenting the relevant part of the loop would generate a lot of unnecessary overhead, since many of the measurements taken while the loop iterates may not be needed.

With this in mind, Dwyer et al. [DPP10] propose a method for determining the *stutter distance* of a loop, meaning the number of times the loop can be unrolled (and instrumented) such that the rest of the (uninstrumented) iterations will not cause undetected violations. This is expressed more formally by letting π be a (non-empty) regular expression capturing all possible

sequences of events in one iteration of the loop being considered, letting d be the number of iterations that can be unrolled, and letting $\Delta(s, \Pi)$ be the set of states in the specification automaton that are reachable from s by following a path in a set of paths Π. They say that a loop *stutters* after d iterations if, for S the set of states in the specification automaton and π^d being π concatenated d times,

$$\forall s \in S : \forall s' \in \Delta(s, \pi^d) : \Delta(s', \pi) = \{s'\}.$$

In other words, after d iterations, any future iteration will always leave the specification automaton in the same state. Given this property, the loop can safely be unrolled and no violations will be missed by not monitoring the remaining loop iterations.

State Estimation

Work done by Bartocci et al. [BGK+13] applies *state estimation* in order to reduce the overhead induced by monitoring. State estimation uses a probabilistic model of the system under scrutiny (with distributions modelling the probability that a given event was observed in a gap during which no observations could be received) to allow monitors with fewer resources (due to an overhead control mechanism removing their resources) to generate a verdict when the input trace is incomplete.

2.10 Explanation of Behaviour Deviations

Explaining deviations of behaviour from what was expected has received a lot of attention in the Model Checking and Fault Localisation communities, but not so much in the Runtime Verification community.

In the work on explanation in this thesis, there is a distinct focus on making the explanation machinery developed accessible to engineers. It would, of course, be possible to give an explanation in terms of the formal semantics of the specification language being used. However, this requires an understanding of the underlying theoretical foundations of the specification language (an explanation in terms of formal semantics requires knowledge of the semantics themselves); our experience with engineers shows such a requirement to count against a tool very heavily when engineers are evaluating the tool's utility. With this in mind, this thesis focuses on delivering an intuitive explanation approach (which includes giving details of the engineering involved), rather than a completely formal one which would be of no use to engineers without further work.

2.10.1 Attention from RV of Software

An approach in Runtime Verification of systems consisting purely of software is to measure the *distance* between a violating trace and the set of traces accepted by the specification that was violated [Reg15]. While this idea has not received much development, it could be extended to determine the closest successful trace, and use that relationship to indicate the differences in the trace that could have avoided violation.

2.10.2 Attention from RV of Cyber-Physical Systems

Despite the lack of work in RV on explanation of property violations in software, there have been contributions recently from the Cyber-Physical Systems (CPS) community. In this setting, one often cares about determining why some model constructed in Simulink [sim] does not satisfy a property written in a specification language suitable for continuous signals, such as Signal Temporal Logic (STL) [MN04].

Given a Simulink model, a specification and a violation, and since the semantics of STL is defined over continuous signals, the explanation problem in the CPS setting often aims to find the interval in which the signal under consideration violated the STL specification. The approach described in [BMM+19] then uses Fault Localisation (see Section 2.10.4) to identify problematic parts of the Simulink model.

2.10.3 Attention from Model Checking

We only enumerate a small subset of the literature on explanation in Model Checking here, the main reason being that Model Checking often differs sufficiently from Runtime Verification to require significant modification, and so we focus on capturing a representative set.

A common approach involves inspecting a set of counterexamples to determine the closest satisfying traces. Given a state space and a property with respect to which it is being checked, a counterexample is a path through the state space that causes a violation of the property. The next step varies; some involve describing differences [GCKS06] and others concentrate on isolating the violating sections of an input signal [FMN15, BBDC+12]. Some work also looks at defining *severity* of satisfaction or violation [DM10].

Finally, one of the most engineer-friendly uses of Model Checking in explanation is that of generating test cases [RH01, ABM98]. In this case, test cases are generated by giving a model checker a set of properties that the relevant system model does not hold. By supplying the model checker with these properties, counterexamples are generated which form test cases. The key aspect of the approach presented in [RH01] is that test cases can be generated with respect to certain coverage specifications, meaning that the approach can be used for safety-critical systems where a certain test coverage is vital.

2.10.4 Explanation as a Fault Localisation Problem

Fault Localisation typically takes certain kinds of data from a program at runtime (for example, data concerning individual system modules, sequences of method calls or sequences of class instantiations) and tries to identify which components of a system could be causing a fault. Common approaches are Spectrum-based [dSCK16] and Model-based [Rei87] Fault Localisation.

The measurement of the effectiveness of these techniques has yielded a number of studies [JH05, AZGvG09, AZG06] and some work has been performed on providing tool-chains to ease the integration of Fault Localisation into software development [GRM+18].

In Spectrum-based Fault Localisation, one of the most notable contributions is Tarantula [JHS01, JH05]. This work first contributes a metric for assigning a *suspiciousness ranking* to program components with respect to failed tests, and then provides an interface for visualisation

of a program along with information recorded during its execution, with the aim of helping software engineers to locate faults. With the same theme of identifying problematic components, a contribution that focuses specifically on finding problematic classes by inspecting sequences of failing method calls is the AMPLE tool [DLZ05].

We highlight that Fault Localisation techniques usually depend on the existence of test cases. For example, spectrum-based approaches involve the collection of certain analytics during test case executions whose outcome is known (ie, it is known whether they pass or fail) in order to find problematic artefacts in the software, such as classes or methods.

An approach that could be placed outside the classification of Spectrum or Model-based Fault Localisation considers error traces and what changes could be made to prune violating extensions [CHM+19]. The idea is that, if a fix can be found that prunes all possible erroneous extensions, the code to be fixed could be regarded as a fault.

Finally, some work looks at measuring the distance between a set of values of given spectra, and a set of values that are known to be acceptable [RR03].

A Problem with Artificial Faults

Despite the high volume of contributions in the Fault Localisation community, one major problem identified has been that the practice of evaluating new algorithms on datasets with artificial faults injected does not give a reliable indication of performance on real faults [PCJ+17]. While the explanation work in this thesis focuses on determining why performance deviated from that described by a specification, we highlight that our experimental setup described in Chapter 7 means that we do not have to inject artificial faults.

2.10.5 Comparing Program Paths

There is some work that considers the comparison of program paths [BL96, RBDL97]. This work differs from approaches in the RV and Model Checking setting in that it looks directly at what happened during execution at the source code level, rather than working with some trace that abstracts the events that occurred at runtime away from the source code.

2.11 Presenting Explanation Results to Engineers

The work that we have summarised so far from the various communities that work on some form of explanation has focused on the theoretical aspect of the problem: *how to go from monitoring results to some information about what went wrong.* While solving this part of the problem is important, we highlight that a second part of the problem must be solved for any contributions to have ultimate utility for software engineers: *how to present information to engineers in a useful way.*

2.11.1 Visualisation in the IDE

The area of RV focusing on presenting monitoring results visually to software engineers is under-developed (since explanation itself is under-developed, this is expected for what is essentially

the follow-up problem). Our contribution, in the form of offline analysis tools, is presented in Chapter 6 and is the most significant effort in this direction of which we are aware.

There have been many contributions from the Software Engineering community on how to visualise data extracted from programs at runtime. The recent focus has been on displaying information collected at runtime inline with the statements that generated it [CLRG19, BMDR13, MMRL16, COL+17, RHV+09]. However, since this work was not done in the formal setting, the emphasis is on presenting results from various kinds of profiling, rather than presenting results of monitoring for formal specifications.

2.11.2 Visualisation of Program Structure and Traces

Visualisation of software has received much attention from the wider Software Engineering community. A notable recent contribution is the notion of a *software city*, described in [RCE+19]. This visualisation involves using static analysis of large software to build a city-like three-dimensional visualisation. Such a visualisation allows software engineers to understand the structure of their software. Further, the JIVE tool [ZJLS16] enables software engineers to visualise runs of Java programs (it also allows some verification).

2.11.3 Development in Industry

An important characteristic of the work presented in this thesis is that it was performed in the industrial context in which it would ultimately be used. Hence, we now highlight work from the RV community that 1) has been performed in close collaboration with industry, leading to decisions on the specification, instrumentation and monitoring approaches that reflected the requirements coming from industry; and 2) directly reflects on the context in which it was performed.

We highlight the work presented in [CP18]. Here, a major challenge faced in this thesis is discussed: applying implementations developed in the RV research context to real-world systems. The implementation that we present in Chapter 6 describes the current status of an ecosystem of tools that are moving from the initial research-oriented environment in which they were built, to a production-ready state for which we can give guarantees about functionality.

2.12 Overview

We have introduced the existing contributions from the Runtime Verification, Software Engineering, Mode Checking and Fault Localisation communities. We have summarised work on specification methods for RV, including temporal logics, rule systems, stream equations and automata. Further, we have highlighted the work on instrumentation, giving particular attention to the lack of such work on instrumentation of Python programs. We have also highlighted the lack of research on explaining property violations by software in the RV community, instead pointing out the contributions from Model Checking and Fault Localisation.

Finally, we have pointed out the importance of the interface between software engineers and theoretical machinery. In particular, over the course of this thesis, while there is work from

the Software Engineering community on developing intuitive interfaces for software engineers to analyse their programs, this effort has not been reciprocated in RV. Hence, this thesis makes significant contributions to translating all of the theory introduced in the first few chapters into a framework that can be used in the software development process.

Chapter 3

A Temporal Logic for Source Code Level Properties

This chapter will develop new a specification language that aims at being easy to use for software engineers and therefore useful as a software development tool. This work was outlined in a paper [DR19b] at SAC-SVT 2019.

Experience with discussing specification languages with software engineers at CERN has shown that, despite the fact that some existing temporal logics are highly expressive with their modal operators, such expressive power is often not needed and can even lead to simple properties being difficult to express. Additionally, whilst a high level of abstraction can make a logic useful for describing behaviour at different levels of granularity, this can also lead to difficulty when maintaining and understanding specifications.

With this in mind, the approach taken in this thesis is to develop a specification language that:

1. Works at a low level of abstraction, preferably even without the need for an instrumentation mapping.

2. Allows expression of properties that engineers are commonly interested in without complex modal operators.

The specification language introduced, called *Control-Flow Temporal Logic* (CFTL), is a linear-time temporal logic designed to be low-level, tightly coupled with the control-flow in a program and free of complex modal operators. Properties captured by CFTL specifications are restricted to single function calls and describe constraints over a mixture of the values held by local variables and timing information obtained by measuring durations of various events.

The benefits of the low level of abstraction of the logic are twofold. Firstly, no instrumentation mapping is required to define the correspondence between the specification and events generated by the program at runtime. This removes the need to maintain two pieces of information separately. The result of this is that the events that occur at runtime which are relevant to the specification can immediately be inferred from the specification itself, which means that the property being expressed can be understood just by looking at the specification. Secondly,

the direct relationship between specifications and control-flow admits a conservative instrumentation algorithm. This results in reduced overhead at runtime because 1) less instrumentation code is being executed; and 2) the monitoring algorithm does not have to spend as much time deciding whether or not measurements taken by instrumentation are relevant.

The advantage of losing complex modal operators lies in 1) simplifying the monitoring process (future-time operators such as *eventually* and even the past-time operators make the monitoring problem challenging); and 2) the ability to more easily express simpler (often more common) properties. Despite the clear loss of expressive power, experience with case studies at CERN has shown that the properties needed to analyse the performance of some real systems (for example, the one used as a case study in Chapter 7) can often be expressed with a less expressive language.

3.1 The Path to Defining Control-Flow Temporal Logic

We begin by giving the steps required to fully define the syntax, semantics, instrumentation strategy and monitoring algorithm for Control-Flow Temporal Logic.

A Model of Programs. Our first step is to take a program P (this thesis focuses on Python programs, but it should be emphasised that this work only assumes a simple imperative language, and not specifically a Python program) and construct its *Symbolic Control-Flow Graph* (SCFG). The SCFG of a program P is a statically-computable directed graph that encodes information about changes to program variables, calls of functions and reachability in control-flow.

A Model of Program Runs. With the definition of a symbolic control-flow graph complete and an algorithm given for computing it for a given program P written in an imperative language that we define, a new kind of trace can be built over which CFTL specifications are evaluated, which we call a *dynamic run*. Dynamic runs have an important property that each such run contains information that indicates parts of the relevant symbolic control-flow graph (and, therefore, source code) that were traversed at runtime. Further, dynamic runs contain source-code level information and allow checking of CFTL specifications.

CFTL Syntax and Semantics. We introduce the syntax of CFTL, enabling software engineers to express properties that should be held by dynamic runs. The syntax of CFTL diverges quite extremely from the conventional temporal logics in an effort to form a more intuitive specification language. Finally, the definition of truth is given for a CFTL specification with respect to a given dynamic run.

The Monitoring Problem and a First Algorithm. A first attempt at a monitoring algorithm using only a CFTL specification and a dynamic run as input is developed. However, the time complexity of this naive approach makes it unsuitable for use on a real-world system. Based on this insight, we turn our attention to instrumentation as a way to optimise the monitoring process with the aim of making it usable in a real-world setting.

Conservative Instrumentation. Given the SCFG of a program and a CFTL specification, a recursive scheme is developed that maps from the CFTL specification to a conservative set of symbolic states in the SCFG. By developing such a scheme, the need for an instrumentation language is avoided and the process of instrumentation is completely automated. Further, instrumentation is used to generate a dynamic run that contains *enough* information. This process is safe, and throws away no information that would be needed to check the formula.

An Efficient Monitoring Algorithm. Finally, the integration of the SCFG into the CFTL semantics is used to develop an efficient monitoring algorithm. The existence of an efficient monitoring algorithm is of particular importance given that CFTL specifications are always universally quantified with at least one quantifier, a characteristic that historically causes a problem for online monitoring approaches in RV.

3.2 A Model of Programs

The statically computable model of a given program discussed so far in this chapter, the *symbolic control-flow graph* (SCFG), will now be introduced by its formal definition and then a recursive scheme that yields the graph $\mathsf{SCFG}(P)$ of a program P.

The program P will be from a subset of the full set of Python programs, generated by the rule *Program* in the grammar in (3.1). Here, x denotes a program variable and f a function symbol.

$$\begin{array}{lcl} \mathit{Program} & := & x = \mathit{expr} \mid \mathit{Program}; \mathit{Program} \mid \\ & & \mathsf{if}\ \mathit{expr}\ \mathsf{then}\ (\mathit{Program})\ \mathsf{else}\ (\mathit{Program}) \mid \\ & & \mathsf{while}\ \mathit{expr}\ \mathsf{do}\ (\mathit{Program}) \mid \mathsf{for}\ x\ \mathsf{in}\ \mathit{iterator}\ \mathsf{do}\ (\mathit{Program}) \\ \mathit{expr} & := & x \mid f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathit{arithExpr} \mid \mathit{boolExpr} \\ \mathit{iterator} & := & \mathsf{range}(\mathit{expr}, \mathit{expr}) \end{array} \tag{3.1}$$

Further, for a program P generated by this grammar, let the set Sym of its symbols be divided into disjoint $\mathsf{VarP}, \mathsf{VarR}, \mathsf{Fun} \subseteq \mathsf{Sym}$; the sets of primitive variable symbols, reference variable symbols and function symbols respectively. The heap is modelled by this division of program symbols into reference and primitive types and concurrency is not considered. There are two motivations for avoiding concurrency:

- Given that this thesis limits its attention to Python, we are bound by the Python interpreter's limitation with respect to concurrency. Principally, Python uses a *global interpreter lock* whose result for programs with no IO is serial execution. Therefore, programs cannot truly execute concurrently.
- Experience shows that it often suffices to describe program behaviour using a small set of program points within single functions, a setting in which considering concurrency does not help.

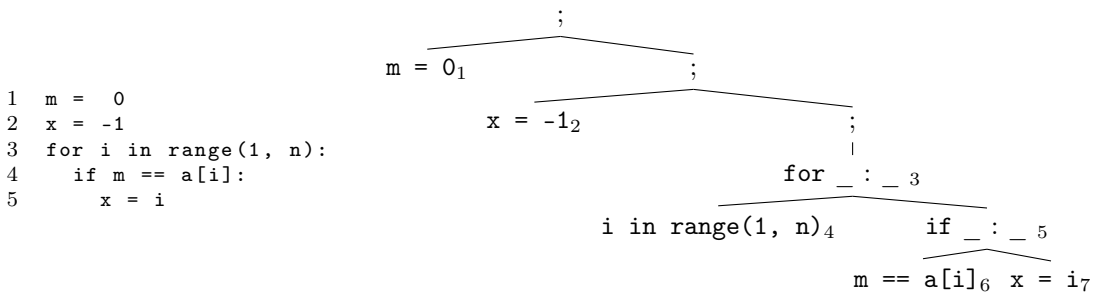


Figure 3.1: The abstract syntax tree of a program with a for-loop and a conditional.

3.2.1 Symbolic States

The notion of a *symbolic state* is used throughout this thesis from the initial development of CFTL all the way through to the explanation machinery presented later. The motivation for such *symbolic states* is to capture the intuition that a given statement in a program (an assignment or function call) results in a change of some quantity measurable at runtime.

If we consider an assignment statement, the immediate quantity to be considered as changed is the target program variable of the assignment. There may also be functions that were called in order to compute the final value assigned. Alternatively, considering a function call statement, the immediate change to take into account is the target function being called, the parameters to which may also have included a function call to compute a final value.

With this motivation in mind, let a *symbolic state* σ be a pair $\langle \rho, m \rangle$ where m is a map from program symbols to statuses $\{\text{changed}, \text{unchanged}, \text{undefined}, \text{called}\}$. The addition of ρ is to prevent symbolic states from being isomorphic (which is easy given the simplicity of their codomain), hence ρ is any uniquely identifying quantity. The quantity chosen here is a *program point*, which is an integer associated with each node in the abstract syntax tree (ie, a parse tree) of a program with respect to the grammar in (3.1). An example of such a unique assignment of program points is given in Figure 3.1; each node in the tree on the right is labelled with an integer. Further examples are given in Figures 3.2 and 3.3. We define the notion of a *program point* more precisely in Section 3.2.2.

Symbolic states are divided into two categories: *standard* symbolic states and *control-flow* symbolic states. For standard symbolic states the program point identifies a node in the program's parse tree that does not contain `if`, `for`, etc. For control-flow symbolic states the program point identifies a node that *does* contain a control-flow keyword. This distinction is useful because SCFGs contain special symbolic states to make control-flow, such as conditionals and loops, more obvious.

3.2.2 A Directed Graph with Symbolic States

Symbolic control-flow graphs are directed graphs whose vertices are symbolic states. With the two categories of symbolic states defined, we now define the notion of a symbolic control-flow graph independently from any notion of a program, and then discuss how one can construct such a graph based on a program written in a given language.

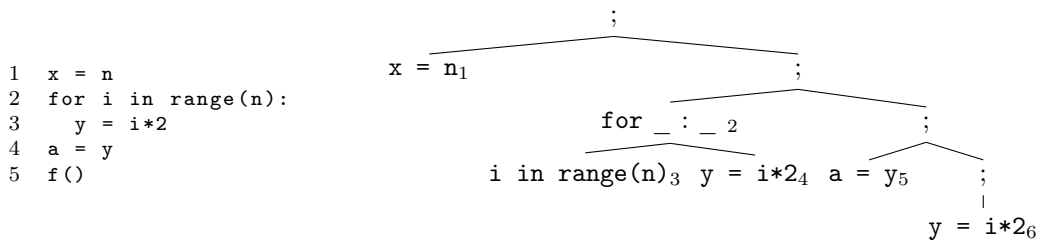


Figure 3.2: The abstract syntax tree of a program with a for-loop and no conditional.

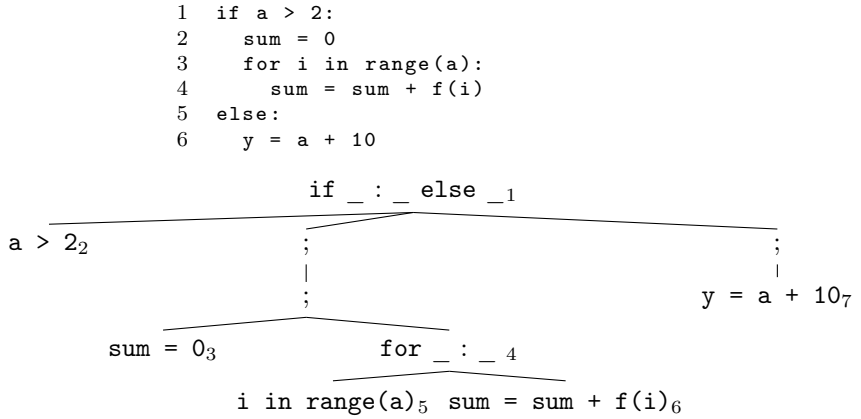


Figure 3.3: The abstract syntax tree of a program with a for-loop nested inside a conditional.

Definition 1. *The symbolic control-flow graph is a directed graph* $\langle V, E, v_s \rangle$ *for a set* V *of symbolic states, a set* $E \subset V \times V$ *of edges and a starting symbolic state* $v_s \in V$*.*

We sometimes abbreviate an edge $\langle \sigma, \sigma' \rangle \in E$ to e. A symbolic state $\sigma \in V$ is *final* if it has no successors, that is, there is no $\sigma' \in V$ such that $\langle \sigma, \sigma' \rangle \in E$. A path π through a symbolic control-flow graph is a sequence of edges $\langle \sigma_1, \sigma_2 \rangle, \langle \sigma_2, \sigma_3 \rangle, \ldots, \langle \sigma_n, \sigma_{n+1} \rangle, \langle \sigma_{n+1}, \sigma_{n+2} \rangle$. Such a path is said to be *complete* if $\sigma_1 = v_s$ and σ_{n+2} is final. We define the *subpath* relation on paths $\pi = e_1 \ldots e_k$ and $\pi' = e'_1 \ldots e'_m$ with $k \leq m$ by $\pi \sqsubset \pi'$ if and only if $e_i = e'_i$ for every $1 \leq i \leq k$.

We now make the connection between a symbolic control-flow graph and a program by 1) labelling each statement in a program P with a unique identifier; and 2) giving a recursive scheme for construction of a graph consistent with Definition 1 based on the program P.

In order to uniquely identify statements in code, for a statement s in a program P, we introduced its *program point*, which is a unique identifier with respect to the program in which s is found. This is demonstrated in Figures 3.1, 3.2 and 3.3. Supposing that the program P consists of the statement s followed by a subprogram P' (hence, $P = s; P'$ for ; a standard delimiter of statements), the program point of the statement s is denoted by $p(s; P')$ and is the integer associated with the node at which s is found in the parse tree of P.

Now, for construction of the symbolic control-flow graph of a program P (using this notion of program points to label symbolic states), we will introduce a recursive scheme by looking at each type of construct in the grammar in (3.1) and giving the scheme for that case. The result is a recursive function $\mathcal{T}$ which takes a symbolic state σ and a program P and returns the set

$$\mathcal{T}(\sigma, x = expr; P) =$$

$$\begin{cases} \{\langle \sigma, \langle p(P), [x \mapsto \text{changed}]\rangle\rangle\} \cup \ \mathcal{T}(\langle p(P), [x \mapsto \text{changed}]\rangle, P) & \mathsf{fn}(expr) = \emptyset \\ \{\langle \sigma, \langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle\rangle\} & \\ \quad \cup \mathcal{T}(\langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle, P) & \text{otherwise} \\ \text{for } x_i \in \mathsf{VarR} \text{ and } f_i \in \ \mathsf{fn}(expr) & \end{cases}$$

Figure 3.4: The definition of $\mathcal{T}$ for simple program statements with no branching.

of edges in a symbolic control-flow graph rooted at σ. Using $\mathcal{T}$, the symbolic control-flow graph of an entire program P itself is given by

$$\mathsf{SCFG}(P) = \langle \overbrace{\{\sigma_1, \sigma_2 \mid \langle \sigma_1, \sigma_2\rangle \in \mathcal{T}(\langle p(P), []\rangle, P)\}}^{V}, \overbrace{\mathcal{T}(\langle p(P), []\rangle, P)}^{E}, \overbrace{\langle p(P), []\rangle}^{v_s}\rangle.$$

Here, the set V of vertices is derived by taking the edge set given by the recursive scheme and extracting the vertices from each edge. The set E of edges is derived simply by applying the recursive scheme, which is now described.

Assignments. For an assignment statement, the symbolic control-flow graph should consist of a vertex in which the target variable of the assignment has been changed and any functions used in the expression have been called. Further, if there are reference variables, they should be considered as changed by any functions used in the expression. To handle this, we introduce $\mathsf{fn}(expr)$, the set of function symbols found in $expr$. We present the construction in Figure 3.4.

Conditionals. In this case, we introduce an intermediate *control-flow* symbolic state to indicate that there is branching in the graph. We do this by using the program point associated with the conditional block's node in the program's parse tree. From here, we recurse on the bodies of each clause P_1 and P_2. We then use a second additional vertex to represent the end of the conditional block. Each clause in the conditional has an edge going from its final vertex to the additional vertex at the end of the conditional block. This models control flowing out of each clause of the conditional and allows it to progress to whatever comes after the conditional. We must also consider the fact that the condition b may contain calls to functions. In this case, these functions must be considered as having been called, and reference variables must be considered as having been changed. We present the construction in Figure 3.5.

While-loops. For while-loops, we introduce two intermediate vertices: one for the beginning of the loop, and one for the end. For the intermediate vertex at the beginning of the loop, there must be an edge leading into the loop body. There must also be a vertex skipping the body completely to go to the intermediate vertex at the end of the loop. This models the fact that the loop condition may never be satisfied, in which case no iterations would ever be completed. Further, the final vertex in the loop body must have an edge leading to the intermediate vertex

$$\begin{aligned}\mathcal{T}(\sigma, \mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2; P_3) = \quad & \{\langle \sigma, \langle p(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), m\rangle\rangle\} \\ & \cup\, \mathcal{T}(\langle p(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), m\rangle, P_1) \\ & \cup\, \mathcal{T}(\langle p(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), m\rangle, P_2) \\ & \cup\, \mathcal{T}(\langle p(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), []\rangle, P_3) \\ & \cup\, \{\langle \sigma', \langle p(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), []\rangle\rangle : \\ & \qquad \sigma' \text{ is final in the subgraph for } P_1 \text{ or } P_2\}\end{aligned}$$

$$\text{where } m = \begin{cases} [x_i \mapsto \text{changed}, f_i \mapsto \text{called}] \text{ for } x_i \in \mathsf{VarR} \text{ and } f_i \in\ \mathsf{fn}(b) & \mathsf{fn}(b) \neq \emptyset \\ [] & \text{otherwise} \end{cases}$$

Figure 3.5: The definition of $\mathcal{T}$ for conditional blocks.

$$\begin{aligned}\mathcal{T}(\sigma, (\mathsf{while}\ b\ \mathsf{do}\ P_1); P_2) = \quad & \{\langle \sigma, \langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), m\rangle\rangle\} \\ & \cup\, \mathcal{T}(\langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), m\rangle, P_1) \\ & \cup\, \{\langle \langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), m\rangle, \langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), []\rangle\rangle\} \\ & \cup\, \mathcal{T}(\langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), []\rangle, P_2) \\ & \cup\, \{\langle \sigma', \langle p(\mathsf{while}\ b\ \mathsf{do}\ P_1), m\rangle\rangle\}\end{aligned}$$

$$\text{where } m = \begin{cases} [x_i \mapsto \text{changed}, f_i \mapsto \text{called}] \text{ for } x_i \in \mathsf{VarR} \text{ and } f_i \in\ \mathsf{fn}(b) & \mathsf{fn}(b) \neq \emptyset \\ [] & \text{otherwise} \end{cases}$$
and σ' is final in the subgraph for P_1

Figure 3.6: The definition of $\mathcal{T}$ for while-loops.

at the beginning of the loop to actually allow looping. We present the construction in Figure 3.6.

For-loops. The translation for for-loops has the slight difference from while-loops because of the necessary change to the loop variable. In this case, the intermediate vertex at the beginning of the loop holds the information that the loop variable has changed. We present the construction in Figure 3.7.

In Figures 3.8 and 3.9 we give examples of programs with their symbolic control-flow graphs. In each case, a Python program is given on the left, and the SCFG constructed using the scheme we have just developed is given on the right.

When SCFGs are drawn we include additional information in the vertices that we draw, such as labelling control-flow vertices according to the type of control-flow they encode. For example, a vertex that represents the beginning of a loop is labelled by loop before the map it contains. Recall that this distinction is possible because each vertex is associated (implicitly, in the examples given) with a program point that identifies the relevant statement/block.

The next step is to take the symbolic control-flow graph $\mathsf{SCFG}(P)$ of a program P and

$$\begin{aligned}\mathcal{T}(\sigma, (\textsf{for } x \textsf{ in } R \textsf{ do } P_1); P_2) = \quad & \{\langle \sigma, \langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), m\rangle\rangle\} \\ & \cup \mathcal{T}(\langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), m\rangle, P_1) \\ & \cup \{\langle\langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), m\rangle, \langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), []\rangle\rangle\} \\ & \cup \mathcal{T}(\langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), []\rangle, P_2) \\ & \cup \{\langle \sigma', \langle p(\textsf{for } x \textsf{ in } R \textsf{ do } P_1), m\rangle\rangle\}\end{aligned}$$

$$\text{where } m = \begin{cases} [x \mapsto \text{changed}, x_i \mapsto \text{changed}, f_i \mapsto \text{called}] & \textsf{fn}(R) \neq \emptyset \\ \quad \text{for } x_i \in \textsf{VarR} \text{ and } f_i \in \textsf{fn}(R) & \\ [] & \text{otherwise} \end{cases}$$

and σ' is final in the subgraph for P_1

Figure 3.7: The definition of $\mathcal{T}$ for for-loops.

```
x = n
for i in range(n):
  y = i*2
a = y
f()
```

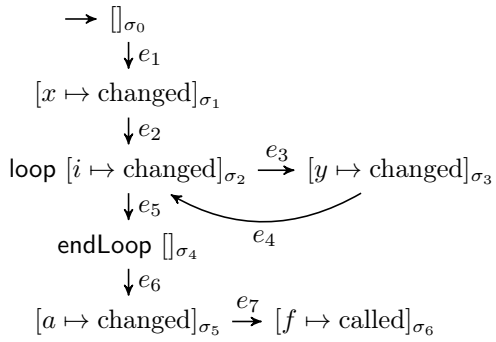


Figure 3.8: A Python program with a for-loop with its symbolic control-flow graph.

consider a run of P as a path through $\textsf{SCFG}(P)$. We will use such a path as a foundation for the new kind of trace that we introduce, where these traces are based on paths, or sets of paths in some cases, through symbolic control-flow graphs with concrete values from the program run and timing information attached.

3.3 A Trace for representing Program Runs

Now that we have a statically-computable structure to represent programs, the next step towards developing CFTL is to introduce a representation of program runs. Such representations will be the *traces* over which CFTL specifications are evaluated. Since CFTL specifications will be able to reason over values held in memory and the time taken by certain events, it is clear that our representation of program runs must encode information about the values generated at runtime and the time at which these values were attained.

```
if a > 2:
  sum = 0
  for i in range(a):
    sum = sum + f(i)
else:
  y = a + 10
```

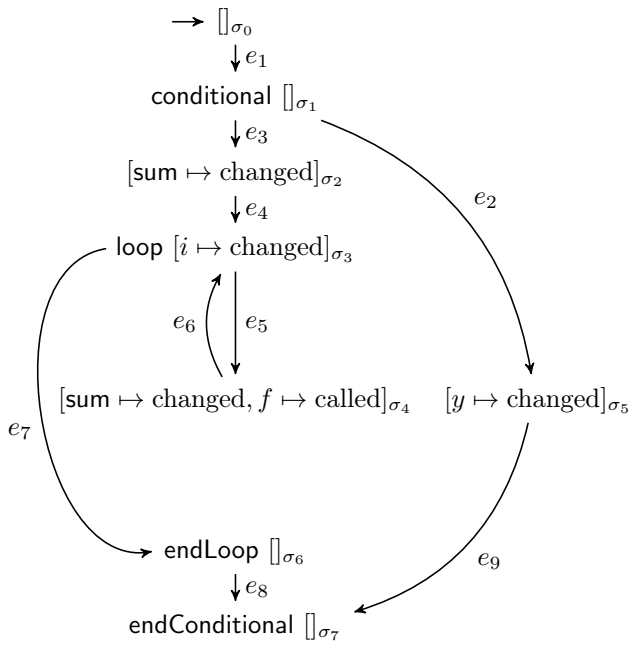


Figure 3.9: A Python program with a conditional and for-loop with its symbolic control-flow graph.

3.3.1 Concrete States

We begin by introducing *concrete states* as instantaneous checkpoints attained during the runtime of a program. The motivation behind introducing these is that we will have structures that describe the state in terms of values held in memory and the time at which that state was attained. By chaining these together, we will build traces.

Formally, concrete states are triples $\langle t, \sigma, \tau \rangle$. We encode time by the timestamp $t \in \mathbb{R}^{\geq}$ and concrete values held by program variables by the map $\tau : \mathsf{Sym} \to \mathsf{Val}$ where Val is the finite set of all values possible at runtime. We also associate each concrete state with a symbolic state σ, which will help when we need to perform instrumentation and make monitoring more efficient. For a concrete state $s = \langle t, \sigma, \tau \rangle$, we denote by $\mathsf{time}(\langle t, \sigma, \tau \rangle)$ the time t at which the concrete state was attained. We abuse notation and write $s(x)$ to mean the value to which τ maps x in s.

Using this definition of a concrete state, the traces over which CFTL operates follow naturally. We define a *dynamic run* as a path through a symbolic control-flow graph with information from the dynamic context attached.

Definition 2. *A dynamic run $\mathcal{D}$ over $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ is a sequence of concrete states $\langle t_1, \sigma_1, \tau_1 \rangle, \ldots, \langle t_n, \sigma_n, \tau_n \rangle$ such that the t_i are strictly increasing and there is a path through $\mathsf{SCFG}(P)$ between every consecutive pair of symbolic states σ_i, σ_{i+1}, ie, $\langle \sigma_1, \sigma_2 \rangle, \ldots, \langle \sigma_{n-1}, \sigma_n \rangle$ can be extended to a path.*

A dynamic run $\langle t_1, \sigma_1, \tau_1 \rangle, \ldots, \langle t_n, \sigma_n, \tau_n \rangle$ is *complete* if $\sigma_1 = v_s$ and σ_n is final in $SCFG(P)$. A dynamic run is *partial* if $\sigma_1 = v_s$ and σ_n is not final. Bearing in mind that Definition 2 allows

```
m =  0
x = -1
for i in range(1, n):
  if m == a[i]:
    x = i
```

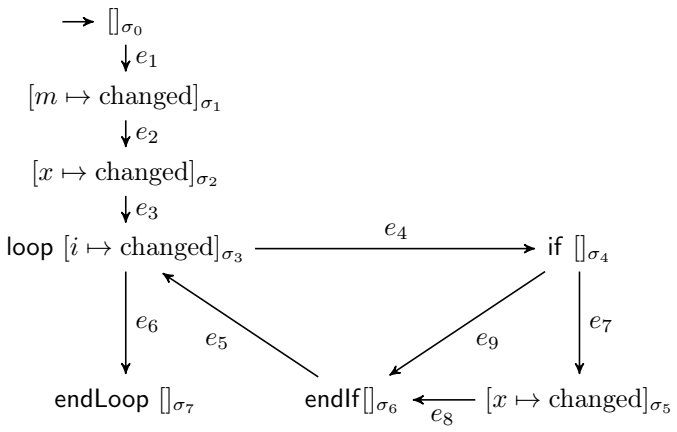


Figure 3.10: A Python program with a for-loop and nested conditional with its symbolic control-flow graph.

for dynamic runs taken from arbitrary places in a symbolic control-flow graph, we will consider only those that are complete or partial. Finally, we denote by $\mathsf{states}(\mathcal{D})$ the set of concrete states occurring during $\mathcal{D}$.

3.3.2 Transitions

We now consider pairs of concrete states taken from a dynamic run and develop properties around them that will allow us to measure and then reason about durations of events found in traces.

For a dynamic run $\mathcal{D}$ with the sequence of concrete states $\langle t_1, \sigma_1, \tau_1\rangle, \ldots, \langle t_n, \sigma_n, \tau_n\rangle$, we call a pair of consecutive concrete states a *transition*. For a transition $tr = \langle t_i, \sigma_i, \tau_i\rangle, \langle t_{i+1}, \sigma_{i+1}, \tau_{i+1}\rangle$, if the only acyclic path between σ_i and σ_{i+1} in the symbolic control-flow graph is of length 1, the transition is said to be *atomic*. A dynamic run is *most-general* if all of its transitions are atomic. The intuition behind this *most-general* criterion is that such a dynamic run would be captured if everything were recorded from a program run (every transition being atomic means that the transition between every pair of consecutive concrete states must represent a single statement being executed).

Since the concrete state at *each end* of a transition has a time at which it was attained, we define the *duration* of the transition as the time elapsed to reach one concrete state from the previous, hence $\mathsf{duration}(tr) = \mathsf{time}(s_{i+1}) - \mathsf{time}(s_i)$. Since transitions model the computation required to move from one state to the next, this notion of duration captures the time taken by such computation. Further properties of a transition $tr = \langle s_i, s_{i+1}\rangle$ include the concrete state in which it originated, $\mathsf{source}(tr) = s_i$, and the concrete state in which it finished, $\mathsf{dest}(tr) = s_{i+1}$. Using this, we capture the intuition that the time at which a transition was attained is the same as the time at which its source state was attained by writing $\mathsf{time}(tr) = \mathsf{time}(\mathsf{source}(tr))$.

To give a final property of transitions, we observe that with our definition of dynamic runs there is no requirement for there to be a single path between the symbolic states associated with consecutive concrete states. Therefore, we denote by $\mathsf{paths}(tr)$ the set of paths through

$\mathsf{SCFG}(P)$ from $\mathsf{source}(tr)$ to $\mathsf{dest}(tr)$.

3.3.3 Examples of Dynamic Runs over SCFGs

As a first example, we consider the SCFG in Figure 3.8 as a basis on which to build dynamic runs. Suppose that the program runs with three iterations of the loop. Then a resulting (most-general) dynamic run with arbitrary timing information and data values could be

$$\begin{gathered}\langle 0.1, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \\ \overbrace{\langle 0.3, \sigma_2, [i \mapsto 0] \rangle}^{\text{loop entry vertex}}, \langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle, \\ \overbrace{\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle}^{\text{loop entry vertex}}, \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle, \\ \overbrace{\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle}^{\text{loop entry vertex}}, \overbrace{\langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle}^{\text{loop exit vertex}}, \\ \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \overbrace{\langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}^{\text{function } f \text{ called}}.\end{gathered} \tag{3.2}$$

From this example it is clear that a dynamic run of a program P is constrained to follow a path through the symbolic control-flow graph $\mathsf{SCFG}(P)$. Another example of a dynamic run that conforms to Definition 2 could be

$$\begin{gathered}\langle 0.1, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \overbrace{\langle 0.3, \sigma_2, [i \mapsto 0] \rangle}^{\text{loop entry vertex}}, \overbrace{\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle}^{\text{loop entry vertex}}, \\ \overbrace{\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle}^{\text{loop entry vertex}}, \overbrace{\langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle}^{\text{loop exit vertex}}, \\ \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \overbrace{\langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}^{\text{function } f \text{ called}}.\end{gathered}$$

in which some concrete states have been thrown away, in particular those that were associated with the symbolic state σ_3. Notice that in this case the dynamic run is no longer most-general, since it is not possible to find a path of length 1 from σ_2 to itself. Further, considering the transition $tr = \langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle$, the set $\mathsf{paths}(tr)$ in fact contains infinitely many paths, since the shortest path from σ_2 to itself is a cycle, meaning further paths can be constructed using arbitrary concatenations of this cycle.

This throwing away of concrete states from a dynamic run is essentially what our instrumentation strategy will be based on. Since a dynamic run is generated by a run of a program, we can take a dynamic run containing only the concrete states that are useful for checking our specification. Whether or not a concrete state is useful can be determined based on its symbolic state. We will discuss this in detail in Chapter 4.

We now give two further examples based on Figures 3.9 and 3.10. Based on Figure 3.9, we

give an example of a most-general dynamic run:

$$\langle 0.1, \sigma_0, []\rangle, \overbrace{\langle 0.15, \sigma_1, []\rangle}^{\text{conditional entry vertex}}, \langle 0.2, \sigma_2, [x \mapsto 0]\rangle,$$
$$\overbrace{\langle 0.21, \sigma_3, [i \mapsto 0]\rangle}^{\text{loop entry vertex}}, \langle 0.5, \sigma_4, [\texttt{sum} \mapsto 1]\rangle,$$
$$\ldots,$$
$$\overbrace{\langle 1.5, \sigma_6, []\rangle}^{\text{loop exit vertex}}, \overbrace{\langle 1.55, \sigma_7, []\rangle}^{\text{conditional exit vertex}}$$

along with an example of a most-general dynamic run based on Figure 3.10:

$$\langle 0.15, \sigma_0, []\rangle, \langle 0.2, \sigma_1, [m \mapsto 0]\rangle, \langle 0.21, \sigma_2, [x \mapsto -1]\rangle,$$
$$\overbrace{\langle 0.25, \sigma_3, [i \mapsto 1]\rangle}^{\text{loop entry vertex}}, \overbrace{\langle 0.27, \sigma_4, []\rangle}^{\text{conditional entry vertex}}, \langle 0.28, \sigma_5, [x \mapsto 0]\rangle,$$
$$\overbrace{\langle 0.3, \sigma_6, []\rangle}^{\text{conditional exit vertex}}, \ldots, \overbrace{\langle 0.41, \sigma_7, []\rangle}^{\text{loop exit vertex}}$$

In both cases we have omitted symbolic states generated by loop iterations to save space.

3.4 Syntax

We now define the structure of formulas in Control-Flow Temporal Logic (CFTL) beginning with the grammar in Figure 3.11. In this grammar, the rule for the non-terminal symbol ϕ is the rule to be applied first. The other non-terminal symbols are then Γ_S, Γ_T, ϕ_M, ϕ_{mixed}, *expr*, $expr_S$, $expr_T$, ϕ_S, ϕ_T, S, T, *cmp* and *arithOp*. In the grammar, non-terminal symbols used in the right-hand-side of rules are highlighted in blue. The rules are organised as follows:

1. After the generation of the specification is started by ϕ, the prefix of the specification that defines quantifiers is generated by repeated applications of ϕ.
2. A boolean combination of atoms is then generated by application of ϕ_M.
3. Individual atoms (which express constraints over quantities extracted from dynamic runs) are then constructed by application of ϕ_{mixed}, ϕ_S and ϕ_T.

We now describe each rule:

- ϕ, being the first rule to be applied, can be recursively applied to generate as many universal quantifiers as are needed. Hence, $\forall$, $\in$ and : are terminal symbols, along with a concrete state variable represented by q and a transition variable represented by t. When this part of the specification is generated, we assume that any letter $a - z$ and $A - Z$ is used (we never have enough quantifiers to require more names of variables than this).

  Further, Γ_S and Γ_T are the non-terminal symbols allowing quantifier predicates to be generated, and ϕ_M is the non-terminal symbol that allows one to generate the *main* part of the specification (the boolean combination of atoms that express constraints over quantities extracted from a dynamic run).

- Γ_S generates expressions used in quantifiers for concrete states. It can generate the predicate $\mathsf{changes}(x)$ (described in Section 3.5.1), where $\mathsf{changes}$ is a terminal symbol along with (,) and x. Further, x represents some program variable. Finally, the predicates $\mathsf{future}(q, \mathsf{changes}(x))$ and $\mathsf{future}(t, \mathsf{changes}(x))$ (also described in (see Section 3.5.1)) can also be generated, with future a terminal symbol. In these cases, q is a terminal symbol representing a variable holding a concrete state and t is a terminal symbol representing a variable holding a transition.

- Γ_T generates expressions used in quantifiers for transitions, and covers similar cases to Γ_S, with the exception of using the calls terminal symbol in order to represent the $\mathsf{calls}(f)$ predicate (described in Section 3.5.1). Here, f is a terminal symbol representing the name of a function.

- ϕ_M enables recursive construction of the inner-most, quantifier-free part of the specification with terminal symbols $\vee$, $\wedge$ and $\neg$ as propositional connectives. Further, the terminal symbol *true* is provided. In order to generate the operands of the propositional connectives, one can either recursively apply ϕ_M or use the non-terminal symbols: ϕ_S (to generate a constraint over a concrete state), ϕ_T (to generate a constraint over a transition) and ϕ_{mixed} (to generate a constraint over multiple concrete states/transitions).

- ϕ_{mixed} enables generation of constraints that 1) compare two quantities or 2) measure the time between two concrete states. The terminal symbols used are $\mathsf{timeBetween}$, (, ,,) and n. $\mathsf{timeBetween}$ refers to an operator that measures the time taken to reach one concrete state from another, and n refers to a real number.

- *expr* generates one of the two non-terminal symbols $expr_S$ and $expr_T$. This is to be used to obtain the concrete states and transitions referred to in constraints that compare multiple quantities. For example, a comparison of a duration with a value held by a concrete state can be generated using these non-terminal symbols.

- $expr_S$ generates the value to which some concrete state (obtained by applying the non-terminal symbol S) maps some program variable x (represented by the terminal symbol x). This value can then be transformed according to *arithOp* and n (as before, a terminal symbol representing a real number). The other terminal symbols in this rule are (and).

- $expr_T$ generates the expression that represents the duration of some transition (obtained by applying the non-terminal symbol T). Hence, $\mathsf{duration}$ is a terminal symbol. The duration represented by $\mathsf{duration}(T)$ can then be transformed according to *arithOp* and n (again, a terminal symbol representing a real number). The other terminal symbols in this rule are again (and).

- ϕ_S generates simple constraints over the value given to a program variable x (represented by the terminal symbol x) by a concrete state (obtained by applying the non-terminal symbol S). The possible constraints are equality to some value v (where v is a terminal symbol representing any string or numerical value) and containment in open and closed

intervals. The terminal symbols in this rule are (,), =, ∈, ,, [and]. n and m are also terminal symbols representing real numbers.

We acknowledge that, given the finite precision of the numbers that can be stored by computers, there is not much point in including the option for both open and closed intervals of the real numbers. However, we include them in order to provide a way to include equality with the interval boundaries without having to write a separate constraint.

- ϕ_T generates simple constraints over the duration of a transition (obtained by applying the non-terminal symbol T). The duration operator is represented by the terminal symbol duration, with other terminal symbols (,), ∈, ,, [and]. n and m are also terminal symbols representing real numbers.

- S generates *concrete state expressions*, to be used anywhere in a specification that refers to a concrete state. For this rule, q, source, dest, source_W, dest_W, next and next_W are terminal symbols, with the usual additions of (,) and ,. The changes(x) predicate over concrete states is represented by the terminal symbol changes. Further, q is a terminal symbol representing a variable holding a concrete state and x is a terminal symbol representing the program variable x. The non-terminal symbols S and T (see below; used to generate transition expressions) are used to obtain concrete states and transitions, respectively, to which the temporal operators source, dest, next, etc, can be applied.

- T generates *transition expressions* in much the same way as S. The exception is the use of t as a terminal symbol representing a variable holding a transition.

- *cmp* generates the comparison operators to be used in constraints that compare multiple quantities.

- *arithOp* generates the terminal symbols × and +. *arithOp* is only used given 1) either the value to which a concrete state maps a program variable, or the duration of a transition; and 2) a real number.

We have now introduced all terminal and non-terminal symbols in the grammar, and given initial intuition as to the meaning of strings that one can generate. In the subsequent sections, we will define some terminology for the structures that can be generated by this grammar (Sections 3.4.1 and 3.4.2), give some criteria for *well-formedness* of specifications that can be generated (Section 3.4.3) and introduce some examples (Section 3.4.4). After that, we will introduce a semantics for these specifications with respect to dynamic runs (Section 3.5).

Finally, we make the following distinction: the abstract object that captures the behaviour of a computational system (a dynamic run, in this case) will from now on be referred to as a *property*. The expression of a property using Control-Flow Temporal Logic will be referred to as a *specification*. When describing parts of a specification, we will still refer to *subformulas*.

3.4.1 Terminology for Atoms

The results of using the ϕ_S, ϕ_T and ϕ_{mixed} rules are known as *atoms*, the set of which in a specification φ we denote by A_φ. We refer to atoms generated by either of the rules ϕ_S or ϕ_T in

$$
\begin{array}{rcl}
\phi & \rightarrow & \forall q \in \Gamma_S : \phi \mid \forall t \in \Gamma_T : \phi \mid \phi_M \\
\Gamma_S & \rightarrow & \mathsf{changes}(x) \mid \mathsf{future}(q, \mathsf{changes}(x)) \mid \mathsf{future}(t, \mathsf{changes}(x)) \\
\Gamma_T & \rightarrow & \mathsf{calls}(f) \mid \mathsf{future}(q, \mathsf{calls}(f)) \mid \mathsf{future}(t, \mathsf{calls}(f)) \\
\phi_M & \rightarrow & \phi_M \vee \phi_M \mid \phi_M \wedge \phi_M \mid \neg\phi_M \mid \phi_S \mid \phi_T \mid \phi_{\mathsf{mixed}} \mid \mathit{true} \\
\phi_{\mathsf{mixed}} & \rightarrow & \mathit{expr}\ \mathit{cmp}\ \mathit{expr} \mid \mathsf{timeBetween}(S, S)\ \mathit{cmp}\ n \\
\mathit{expr} & \rightarrow & \mathit{expr}_S \mid \mathit{expr}_T \\
\mathit{expr}_S & \rightarrow & S(x) \mid S(x)\ \mathit{arithOp}\ n \\
\mathit{expr}_T & \rightarrow & \mathsf{duration}(T) \mid \mathsf{duration}(T)\ \mathit{arithOp}\ n \\
\phi_S & \rightarrow & S(x) = v \mid S(x) \in (n, m) \mid S(x) \in [n, m] \\
\phi_T & \rightarrow & \mathsf{duration}(T) \in (n, m) \mid \mathsf{duration}(T) \in [n, m] \\
S & \rightarrow & q \mid \mathsf{source}(T) \mid \mathsf{dest}(T) \mid \mathsf{source}_W(T) \mid \mathsf{dest}_W(T) \mid \mathsf{next}(S, \mathsf{changes}(x)) \mid \\
 & & \mathsf{next}(T, \mathsf{changes}(x)) \mid \mathsf{next}_W(S, \mathsf{changes}(x)) \mid \mathsf{next}_W(T, \mathsf{changes}(x)) \\
T & \rightarrow & t \mid \mathsf{next}(S, \mathsf{calls}(f)) \mid \mathsf{next}(T, \mathsf{calls}(f)) \mid \mathsf{next}_W(S, \mathsf{calls}(f)) \mid \mathsf{next}_W(T, \mathsf{calls}(f)) \\
\mathit{cmp} & \rightarrow & < \mid > \mid = \\
\mathit{arithOp} & \rightarrow & \times \mid +
\end{array}
$$

Figure 3.11: Syntax of CFTL.

the grammar as *normal* atoms. We refer to atoms generated by the rule ϕ_{mixed} as *mixed* atoms.

The left and right-hand side of a mixed atom are known as *expressions*. We denote by $\mathsf{exprs}(\varphi)$ the set of all expressions in φ. Further, for a mixed atom α we denote the left-most expression by $\mathsf{lhs}(\alpha)$ and the right-most by $\mathsf{rhs}(\alpha)$. For an atom $\alpha \in A_\varphi$, we denote by $\mathsf{vars}(\alpha)$ the set of variables (if α is normal, a singleton) from quantification used by α. Further, for a term t generated by S or T, we denote by $\mathsf{var}(t)$ the single variable from which t is derived. There is no constraint on the variables used on different sides of the same atom, that is, one can relate quantities derived from different states/transitions.

3.4.2 Terminology for Temporal Operators

Given an atom $\alpha \in A_\varphi$, one must be able to *break apart* that atom and look at the sequence of temporal operators used to form it. In terms of the grammar in Figure 3.11, this equates to determining the sequences of applications of the rules S and T.

For a normal atom, there is one sequence of temporal operators; for a mixed atom, there are two. One can construct such a sequence, which we call a *composition sequence* and denote by $\mathsf{composition}(\alpha)$, by considering the parse tree of the atom with respect to the grammar in Figure 3.11. We give a recursive construction in Figure 3.12, in which $+$ denotes sequence concatenation and the notation $\mathsf{rule}(\mathsf{rule}')$ denotes a term generated by first applying rule, and

$$\begin{aligned}
\mathsf{composition}(\mathit{expr}_S(S)) &:= (\mathit{expr}_S) + \mathsf{composition}(S) \\
\mathsf{composition}(\phi_S(S)) &:= (\phi_S) + \mathsf{composition}(S) \\
\mathsf{composition}(S(S')) &:= (S) + \mathsf{composition}(S') \\
\mathsf{composition}(T(T')) &:= (T) + \mathsf{composition}(T')
\end{aligned}$$

Figure 3.12: A part of the recursive construction for the composition sequence of an atom.

$$\begin{aligned}
&\mathsf{composition}(\mathsf{dest}(\mathsf{next}(s, \mathsf{calls}(f)))(x) < 10) \\
= \;& ((\ .\ )(x) < 10) + \mathsf{composition}(\mathsf{dest}(\mathsf{next}(s, \mathsf{calls}(f)))) \\
= \;& ((\ .\ )(x) < 10, \mathsf{dest}(\ .\ )) + \mathsf{composition}(\mathsf{next}(s, \mathsf{calls}(f))) \\
= \;& ((\ .\ )(x) < 10, \mathsf{dest}(\ .\ ), \mathsf{next}(\ .\ , \mathsf{calls}(f))) + \mathsf{composition}(s) \\
= \;& ((\ .\ )(x) < 10, \mathsf{dest}(\ .\ ), \mathsf{next}(\ .\ , \mathsf{calls}(f)), s)
\end{aligned}$$

Figure 3.13: Computing the composition sequence of an atom.

then substituting the non-terminal symbol in that rule for rule'. In Figure 3.13, we give an example of $\mathsf{composition}(\alpha)$ being computed using the recursive definition.

The main reason for defining the *composition sequence* of an atom or expression is to generate a sequence of components used to construct the atom or expression. This sequence of components allows us to extract the temporal operator part of the atom. That is, for a normal atom or expression ψ, we denote by $\mathsf{temp}(\psi)$ the *temporal operator part*. This is obtained by computing the composition sequence and removing the first element. The result of the temp function can always be generated by either the S or T rules in the syntax given in Figure 3.11. As an example, letting $\alpha = \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$, we have

$$\mathsf{temp}(\alpha) = \mathsf{next}(q, \mathsf{calls}(f)).$$

Finally, the difference between next and next_W will be explained when we define the semantics, but for now it suffices to say that next_W is a *weak* version of next. This distinction is only present for complete dynamic runs; when we consider semantics for partial dynamic runs, there will be no distinction.

3.4.3 Well-formedness

Once a specification is built using the grammar, we decide whether it belongs to CFTL based on well-formedness criteria. We say that a CFTL specification generated by the grammar in Figure 3.11 is *well-formed* with respect to a dynamic run if:

1. It is in prenex form, that is, it has a sequence of quantifiers at the beginning, and then an inner part with no quantifiers.

2. Every quantifier depends on the previous. This means that, for the quantifier at position $i > 0$, the term substituted for Γ_S or Γ_T in the construction must use the variable from

the quantifier at position $i - 1$.

3. Every variable in the inner part of the specification is bound by a quantifier, that is, there are no free variables.

4. The inner part of the specification is *well-typed*, that is, measurements performed by operators, comparisons of types and arithmetic performed over measurements make sense.

5. Negation is propagated through to atoms. Any specification for which this is not the case can be rewritten using the DeMorgan laws.

Criterion 1 constrains specifications to a form for which monitoring is known to be efficient and for which a conservative instrumentation strategy can be developed. In particular, if specifications are constrained to have all quantifiers on the left, then monitoring can consist only of computing the set of bindings and evaluating a formula tree at each of these bindings (see Chapter 4).

Criterion 2 rules out specifications that place nonsensical constraints that are extremely hard, if not impossible, for a program to satisfy.

Criterion 3 ensures that there are no uninterpreted variables in the specification and that specifications do not require any further information in the form of an interpretation.

Criterion 4 ensures that the property captured by the specification can actually be checked. For example, if a specification were to attempt to measure the value to which a transition mapped a program variable, given that transitions have no notion of mapping, this specification would be impossible to check. A concrete example of such a situation is $\forall c \in \mathsf{calls}(f) : c(x) < 10$, which can be generated by our grammar but which also treats c as a concrete state (attempting to obtain the value to which it maps the program variable x), despite it being a transition.

Criterion 5 is to reduce the number of cases we have to deal with when defining the monitoring algorithms.

Checking these criteria would be straightforward using the parse tree of a specification with respect to the grammar (and by inspecting the relevant dynamic run), but the implementation presented in Chapter 6 involves construction of CFTL specifications using a Python library in which specification that are not well-formed cannot be constructed.

3.4.4 Examples of Specifications

We give some examples of specifications in Control-Flow Temporal Logic along with their natural language interpretations. We define the precise meaning of these specifications when we develop the semantics in Section 3.5. While in Section 3.4 we refer to rules generating expressions such as timeBetween and duration, in Section 3.5 we define how these expressions can be interpreted in the frame of dynamic runs.

We remark that, since CFTL reasons over only single function calls and does not consider interprocedural or concurrency properties, we give properties that would be used to describe the behaviour of code in a single scope. The examples are as follows:

1. "The calls to function f take less than 5 time units" can be expressed by

$$\forall t \in \mathsf{calls}(f) : \mathsf{duration}(t) \in (0, 5).$$

In this case, for a given dynamic run, the quantifier-free part of the specification $\mathsf{duration}(t) \in (0, 5)$ is checked for each t in the set of transitions found in the dynamic run that satisfy the predicate $\mathsf{calls}(f)$ which will be defined in this chapter.

In particular, for each transition bound to t by quantification, the duration will be measured (using the definition given previously) and compared to the open interval $(0, 5)$.

2. "All calls to function f leave the value of x unchanged" can be expressed by

$$\forall t \in \mathsf{calls}(f) : \mathsf{source}(t)(x) = \mathsf{dest}(t)(x).$$

This specification contains a *mixed* atom $\mathsf{source}(t)(x) = \mathsf{dest}(t)(x)$ of which both sides use the same transition. To evaluate this specification, the inner part will be checked at transitions satisfying the $\mathsf{calls}(f)$ predicate. The check will involve measuring the value of x at either side of each of these transitions and performing the equality check.

3. "Whenever x is changed it is not zero" can be expressed by

$$\forall q \in \mathsf{changes}(x) : \neg(q(x) = 0).$$

4. "Whenever x changes, its value remains unchanged until the next call of f", i.e. f always sees every change to x, can be expressed by

$$\forall q \in \mathsf{changes}(x) : q(x) = \mathsf{source}(\mathsf{next}(q, \mathsf{calls}(f)))(x).$$

This specification uses one of CFTL's few temporal operators, $\mathsf{next}(q, \mathsf{calls}(f))$. The intuition here is to start from the state q that satisfies the predicate $\mathsf{changes}(x)$ and iterate forward through the dynamic run given to find the next transition tr that is a call to f. This will be shown formally in the semantics, but for now it suffices to say that a transition tr is known to be a call of a function f if the symbolic state σ associated with $\mathsf{dest}(tr)$ has $\sigma(f) = \text{called}$.

5. Finally, "whenever x changes, if its value is in $[0, 5)$, then all future calls to f should take units of time in $(0, 10)$" can be expressed (using $[0, 5) = (0, 5) \cup [0, 1]$) by

$$\begin{aligned}
&\forall q \in \mathsf{changes}(x) : \\
&\forall t \in \mathsf{future}(q, \mathsf{calls}(f)) : \\
&\qquad (q(x) \in (0, 5) \vee q(x) \in [0, 1]) \implies \mathsf{duration}(t) \in (0, 10).
\end{aligned}$$

This specification introduces multiple quantification. It is well-formed because the second quantifier uses q from the quantifier immediately before it.

3.5 Semantics

We now begin the process of defining what it means for a dynamic run $\mathcal{D}$ to *satisfy* a CFTL specification φ, that is, for it to hold the property that the specification φ captures. The first semantics that we introduce will be for complete dynamic runs, which are those associated with a path through a symbolic control-flow graph from its the start vertex to a final vertex. We will later introduce a semantics for partial dynamic runs once we have the motivation. To develop this initial semantics we need three things:

1. A relation $\vdash$ (the *quantifier relation*, see Section 3.5.1) which will determine which concrete states and transitions from a dynamic run should be identified by quantification.

2. A function eval (see Section 3.5.3) which will take a dynamic run, a state or transition expression (generated by the rules S and T in the syntax of CFTL in Figure 3.11) and a map from variables to concrete states/transitions. The result will be a unique concrete state or transition from the dynamic run that is needed give a value to a part of a specification.

3. A relation $\models$ (see Section 3.5.4) which will be used to decide satisfaction using the $\vdash$ relation and eval function. This decision will take place using the information obtained by the eval function.

The intuition here is that the $\vdash$ relation will identify the concrete states/transitions in a dynamic run at which the inner-most, quantifier-free part of the specification should be checked. For each concrete state or transition expression in the specification (such as, for example, $\mathsf{next}(q, \mathsf{calls}(f))$), the eval function will be used to determine the unique concrete state or transition in the dynamic run from which information should be taken to decide whether the constraint placed holds. It then remains to decide whether the constraints expressed by atoms hold. For example, if a specification contains the atom $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$, the eval function will first be used to determine the transition (pair of concrete states) that are identified by the transition expression $\mathsf{next}(q, \mathsf{calls}(f))$. With the relevant transition identified, the $\models$ relation will then be responsible for determining whether the duration of this transition is less than 1.

3.5.1 The Quantifier Relation

Figure 3.14 gives the $\vdash$ relation which takes the form $\mathcal{D}, q \vdash \phi$ or $\mathcal{D}, tr \vdash \phi$ for a concrete state q or transition tr, and predicate ϕ generated by rule Γ_S or Γ_T in Figure 3.11. We often omit the dynamic run and simply write $q \vdash \phi$ or $tr \vdash \phi$. Further, we also often write $q \in \Gamma_S$ or $tr \in \Gamma_T$ to mean $q \vdash \Gamma_S$ and $tr \vdash \Gamma_T$ respectively. Since the evaluation of satisfaction of some predicates depends on some other value, for example for $\mathsf{future}(s, \mathsf{changes}(x))$ it depends on s, we write $\Gamma(s)$ (if we are not specifying the predicate) to emphasise dependence and specify the value that will be used when deciding satisfaction of the predicate.

To demonstrate $\vdash$, we now show the dynamic run from (3.2) (see page 62) with highlighting to demonstrate which states would be identified by the quantifier $\forall q \in \mathsf{changes}(y)$. Recall that

$$
\begin{array}{lll}
\mathcal{D}, \langle t, \sigma, \tau \rangle & \vdash & \mathsf{changes}(x) \text{ iff } \sigma(x) = \text{changed} \\
\mathcal{D}, q & \vdash & \mathsf{future}(s, \mathsf{changes}(x)) \text{ iff } \mathsf{time}(q) > \mathsf{time}(s) \text{ and } \mathcal{D}, q \vdash \mathsf{changes}(x) \\
\mathcal{D}, tr & \vdash & \mathsf{calls}(f) \text{ iff for every path } \pi \in \mathsf{paths}(tr) : \\
 & & \quad \text{there is some } \langle \sigma_1, \sigma_2 \rangle \in \pi \text{ such that } \sigma_2(f) = \text{called} \\
\mathcal{D}, tr & \vdash & \mathsf{future}(s, \mathsf{calls}(f)) \text{ iff } \mathsf{time}(tr) > \mathsf{time}(s) \text{ and } \mathcal{D}, tr \vdash \mathsf{calls}(f) \\
\mathcal{D}, tr & \vdash & \mathsf{changes}(x) \text{ iff false} \\
\mathcal{D}, tr & \vdash & \mathsf{future}(s, \mathsf{changes}(x)) \text{ iff false} \\
\mathcal{D}, q & \vdash & \mathsf{calls}(x) \text{ iff false} \\
\mathcal{D}, q & \vdash & \mathsf{future}(s, \mathsf{calls}(f)) \text{ iff false}
\end{array}
$$

Figure 3.14: The $\vdash$ relation for the CFTL semantics.

$\sigma_3(y) = \text{changed}$.

$$
\begin{gathered}
\langle 0, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \\
\langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \overbrace{\langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle}^{\text{selected}}, \\
\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \overbrace{\langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle}^{\text{selected}}, \\
\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle, \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle, \\
\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle.
\end{gathered}
$$

If we instead consider the quantifier $\forall t \in \mathsf{calls}(f)$, recalling that $\sigma_6(f) = \text{called}$, the selection by this predicate would be as follows

$$
\begin{gathered}
\langle 0, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \\
\langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle, \\
\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle, \\
\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle, \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle, \\
\overbrace{\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}^{\text{selected}}.
\end{gathered}
$$

where a transition was selected, since the predicate $\mathsf{calls}(f)$ is over transitions.

3.5.2 Bindings

With the $\vdash$ relation defined, allowing the determination of concrete states or transitions relevant to quantifiers that will ultimately be assigned to the variables used in specifications, we now introduce the machinery that we will need to define the *temporal operators* provided by CFTL. These operators take the concrete states and transitions provided by quantifiers and search forwards through the dynamic run over which the CFTL specification is being evaluated.

$$\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n) =$$

$$\begin{cases} \bigcup_{s_1 \vdash \Gamma_1} \mathsf{bindings}^*_{\varphi}(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n) & n > 1 \\ \{[q_1 \mapsto s_1] : s_1 \vdash \Gamma_1\} & n = 1 \end{cases}$$

$$\mathsf{bindings}^*_{\varphi}(\mathcal{D}, s_k, \forall q_{k+1} \in \Gamma_{k+1} : \cdots : \forall q_n \in \Gamma_n) =$$

$$\bigcup_{s_{k+1} \vdash \Gamma_{k+1}(s_k)} \{[q_k \mapsto s_k] \dagger \beta' : \beta' \in \mathsf{bindings}^*_{\varphi}(\mathcal{D}, s_{k+1}, \forall q_{k+2} \in \Gamma_{k+2} : \cdots : \forall q_n \in \Gamma_n)\}$$

$$\mathsf{bindings}^*_{\varphi}(\mathcal{D}, s_{n-1}, \forall q_n \in \Gamma_n) = \{[q_{n-1} \mapsto s_{n-1}, q_n \mapsto s_n] : s_n \vdash \Gamma_n(s_{n-1})\}$$

Figure 3.15: A recursive construction for $\mathsf{bindings}_{\varphi}$.

Further, the temporal operators require structures that allow us to pass concrete states/-transitions captured by quantifiers to the parts of a specification that use the relevant variables. Such structures are known as *bindings* and are maps $[q_i \mapsto s_i]$ from variables to concrete states/-transitions. We usually denote bindings by β.

For a CFTL specification $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi$, we call a binding *complete* if its domain contains every q_i for $1 \leq i \leq n$ and *partial* otherwise. We require that partial bindings have a non-empty domain. For two bindings β, β', we define the $\sqsubset$ relation by

$$\beta \sqsubset \beta' \iff \mathsf{dom}(\beta) \subsetneq \mathsf{dom}(\beta') \wedge \forall q_i \in \mathsf{dom}(\beta) : \beta(q_i) = \beta'(q_i).$$

Finally, for a dynamic run $\mathcal{D}$ and CFTL specification $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi$, we define the function $\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n)$ that gives the set of *complete* bindings derived from $\mathcal{D}$. $\mathsf{bindings}_{\varphi}$ is defined recursively in Figure 3.15. Notice that the relation $\vdash^*$ is used when selecting concrete states or transitions to be used to extend bindings. Finally, $\mathsf{bindings}_{\varphi}$ is computed by Algorithm 1.

We now prove the correctness of Algorithm 1, which will help us to prove later that our initial monitoring algorithm is correct.

Theorem 1. *Algorithm 1 is correct.*

Proof. We assume a CFTL specification with quantifiers $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$.

Correctness in this case means that the variable C in Algorithm 1 holds precisely the same bindings as those generated in the construction in Figure 3.15. We first highlight that both the algorithm and the recursive construction have a mechanism to ensure that only complete bindings are included in the final set. In particular, the algorithm computes M, the set of all bindings (whether complete or not), and then returns C, which is the set of only complete bindings taken from M. In the recursive construction, only complete bindings can be generated. This can be seen by realising that there is a special case in the construction for the final quantifier, in which case a set of maps is returned for this quantifier. Since the definition recurses until this base case, there can be no bindings in the final set that are not complete.

We now prove that each binding found in either $\mathsf{bindings}_{\varphi}(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n)$ or

Algorithm 1 Compute the set of complete bindings given a dynamic run $\mathcal{D}$ and the quantifiers $\forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n$.

```
1: M ← {}                                                    ▷ Empty set of bindings
2: C ← {}      ▷ Empty set of bindings - will contain the set of complete bindings at the end
3: prev ← ⟨0, [], []⟩                                        ▷ To store the previous state
4: for concrete state curr ∈ D do
5:    %Handle the cases where a new binding should be generated
6:    %New bindings are generated if the state/transition is in Γ_1
7:    if curr ∈ Γ_1 then
8:       M ← M ∪ {[s_1 ↦ curr]}
9:    if ⟨prev, curr⟩ ∈ Γ_1 then
10:      M ← M ∪ {[s_1 ↦ ⟨prev, curr⟩]}
11:   %Bindings are extended if the state/transition is in Γ_i for i > 1
12:   for β = [s_1 ↦ v_1, ..., s_k ↦ v_k] in M where k < n do
13:      if curr ∈ Γ_{k+1}(v_k) then
14:         M ← M ∪ {β † [s_{k+1} ↦ curr]}
15:      if ⟨prev, curr⟩ ∈ Γ_{k+1}(v_k) then
16:         M ← M ∪ {β † [s_{k+1} ↦ ⟨prev, curr⟩]}
17:   prev ← curr
18: %Check for complete bindings - we just need those
19: C ← {β : β ∈ M such that β is complete}
20: return C
```

C must be contained by the other set. We begin by taking a binding β from the set C constructed by the algorithm. If β had been constructed by the algorithm, then $q_1 \in \mathsf{dom}(\beta)$ (since no empty bindings can be constructed). If $\mathsf{dom}(\beta) = \{q_1\}$ (the CFTL specification has a single quantifier), then $\beta(q_1)$ satisfied Γ_1. In this case, the recursive construction must select $\beta(q_1)$. If $\mathsf{dom}(\beta)$ contains more than q_1, then it must have been extended. The algorithm extends bindings by checking whether a concrete state curr or transition $\langle \mathsf{prev}, \mathsf{curr} \rangle$ found satisfies the correct predicate. If this is the case, then a previous binding found (that has not already been sufficiently extended) can be extended. Notice that, because of the for-loop in the algorithm, we are implicitly only checking for concrete states or transitions in the future that can be used to extend a binding formed in the past. This behaviour is reflected by the recursive construction with the emphasis of the dependence of the predicate denoted by $\Gamma_{k+1}(s_k)$. Ultimately, since the extension performed by the algorithm is equivalent to that performed by the recursive construction, we have $C \subset \mathsf{bindings}_\varphi(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n)$.

We now prove in the other direction. Consider a binding $\beta \in \mathsf{bindings}_\varphi(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n)$. We again suppose that $\mathsf{dom}(\beta) = \{q_1\}$. In the recursive construction, this can only be generated by the special case for single quantifiers. The check performed here is mirrored by the algorithm, which first generates maps with only a single variable in their domain. If $\mathsf{dom}(\beta)$ contains more than just q_1, then it must have been extended during the recursion. To achieve extension, for each q_k, the recursive construction finds q_{k+1} with $q_{k+1} \vdash \Gamma_{k+1}(q_k)$. Here, the predicate Γ_{k+1} depends on the value of q_k taken from the binding being extended, meaning that any q_{k+1} found will be later in time than q_k. This is reflected in the algorithm, which only extends bindings if a concrete state/transition is found in the

$$\langle 0, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle,$$
$$\langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \underbrace{\langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle}_{q},$$
$$\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle,$$
$$\underbrace{\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle}_{q}, \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle,$$
$$\underbrace{\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}_{t}$$

$$\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall q \in \mathsf{changes}(i) : \forall t \in \mathsf{future}(q, \mathsf{calls}(f)))$$
$$= \left\{ \begin{array}{l} [q \mapsto \langle 0.6, \sigma_3, [...] \rangle, t \mapsto \langle 1.7, \sigma_5, [...] \rangle, \langle 2.5, \sigma_6, [...] \rangle], \\ [q \mapsto \langle 1.5, \sigma_2, [...] \rangle, t \mapsto \langle 1.7, \sigma_5, [...] \rangle, \langle 2.5, \sigma_6, [...] \rangle] \end{array} \right\}$$

Figure 3.16: An example of how the set of bindings would be computed given a CFTL specification and a dynamic run.

future of all concrete states/transitions that form a binding. Ultimately, since the extension performed by the recursive construction is equivalent to that performed by the algorithm, we have $\mathsf{bindings}_{\varphi}(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n) \subset C$.

Hence, $\mathsf{bindings}_{\varphi}(\mathcal{D}, s_1, \forall q_2 \in \Gamma_2 : \cdots : \forall q_n \in \Gamma_n) = C$, and the result follows. □

We conclude our introduction of the $\mathsf{bindings}_{\varphi}$ function with an example in Figure 3.16. In this case, we compute the set of bindings generated with respect to the given dynamic run according to a CFTL specification with quantifiers $\forall q \in \mathsf{changes}(i) : \forall t \in \mathsf{future}(q, \mathsf{calls}(f))$.

3.5.3 The eval Function

Given a dynamic run and a binding, we now introduce a way to determine the concrete state or transition in the dynamic run that a given concrete state or transition expression identifies. The mechanism we introduce is the eval function, defined partly in Figure 3.17. The remaining cases are left out because they can be obtained simply by replacing null results with null_W results.

This function takes 1) a binding β generated by the quantifiers in a CFTL specification and 2) either a concrete state expression (generated by the non-terminal symbol S in the grammar in Figure 3.11) or transition expression (generated by the non-terminal symbol T in the grammar). Hence, the definition of the function must have cases for each concrete state/transition expression that can be generated.

It first refers to the base case of such expressions (variables from quantifiers holding either concrete states or transitions) using q for concrete states and tr for transitions. These base cases of the eval function consist of determining the concrete state or transition to which the binding β sends the variable given.

It then refers to the other cases, which are $\mathsf{source}(T)$ and $\mathsf{dest}(T)$ for some transition expression T, along with $\mathsf{next}(X, \mathsf{changes}(x))$ and $\mathsf{next}(X, \mathsf{calls}(f))$ for some expression X that is either a concrete state expression or a transition expression (generated by S or T in the grammar,

$$
\begin{array}{lcl}
\mathsf{eval}(\mathcal{D}, \beta, q) & = & \beta(q) \\
\mathsf{eval}(\mathcal{D}, \beta, tr) & = & \beta(tr) \\
\mathsf{eval}(\mathcal{D}, \beta, \mathsf{source}(T)) & = & \text{if } \mathsf{eval}(\mathcal{D}, \beta, T) \notin \{\mathsf{null}, \mathsf{null}_W\} \text{ then } \mathsf{source}(\mathsf{eval}(\mathcal{D}, \beta, T)) \\
 & & \text{otherwise } \mathsf{null} \\
\mathsf{eval}(\mathcal{D}, \beta, \mathsf{dest}(T)) & = & \text{if } \mathsf{eval}(\mathcal{D}, \beta, T) \notin \{\mathsf{null}, \mathsf{null}_W\} \text{ then } \mathsf{dest}(\mathsf{eval}(\mathcal{D}, \beta, T)) \\
 & & \text{otherwise } \mathsf{null} \\
\mathsf{eval}\,(\mathcal{D}, \beta, \mathsf{next}(X, \mathsf{changes}(x))) & = & \text{if } \mathsf{eval}(\mathcal{D}, \beta, X) \notin \{\mathsf{null}, \mathsf{null}_W\} \text{ then} \\
 & & \left(\begin{array}{l} q \text{ if there is a } q \text{ such that:} \\ \quad \mathsf{time}(q) > \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X)) \\ \quad \text{and } \mathcal{D}, q \vdash \mathsf{changes}(x) \\ \quad \text{and (there is no } q' \text{ such that:} \\ \qquad \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X)) < \mathsf{time}(q') < \mathsf{time}(q) \\ \qquad \text{and } \mathcal{D}, q' \vdash \mathsf{changes}(x)) \\ \text{otherwise } \mathsf{null} \end{array} \right) \\
 & & \text{otherwise } \mathsf{null} \\
\mathsf{eval}\,(\mathcal{D}, \beta, \mathsf{next}(X, \mathsf{calls}(f))) & = & \text{if } \mathsf{eval}(\mathcal{D}, \beta, X) \notin \{\mathsf{null}, \mathsf{null}_W\} \text{ then} \\
 & & \left(\begin{array}{l} tr \text{ if there is a } tr \text{ such that:} \\ \quad \text{(if } \mathcal{D}, \mathsf{eval}(\mathcal{D}, \beta, X) \vdash \mathsf{calls}(f) \text{ then} \\ \quad\quad \mathsf{time}(tr) > \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X)) \\ \quad \text{otherwise } \mathsf{time}(tr) \geq \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X))) \\ \quad \text{and } \mathcal{D}, tr \vdash \mathsf{calls}(f) \\ \quad \text{and (there is no } tr' \text{ such that:} \\ \qquad \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X)) < \mathsf{time}(tr') < \mathsf{time}(tr) \\ \qquad \text{and } \mathcal{D}, tr' \vdash \mathsf{calls}(f)) \\ \text{otherwise } \mathsf{null} \end{array} \right) \\
 & & \text{otherwise } \mathsf{null}
\end{array}
$$

Figure 3.17: The eval function for the CFTL semantics.

respectively). The use of T is to indicate that the expression expected was generated by the rule T in the grammar in Figure 3.11. Ultimately, these cases involve recursion on the concrete state or transition expressions held by the expression given.

We now demonstrate the eval function using a dynamic run over the SCFG from Figure 3.8. We begin with the simple CFTL specification

$$\forall q \in \mathsf{changes}(a) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5).$$

By inspecting the quantifier-free part of the specification, we see that the single atom

$$\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)$$

has the transition expression $T = \mathsf{next}(X, \mathsf{calls}(f))$ and the concrete state expression $X = q$. Based on this deconstruction of the specification, we first show how the $\vdash$ relation identifies

the concrete states that are relevant for its quantifier, and then how the eval function would identify the next call to f. The result (with sections of the dynamic run annotated accordingly) is as follows:

$$\begin{gathered}\langle 0, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \\ \langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle, \\ \langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle, \\ \langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle, \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle, \\ \overbrace{\underbrace{\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}_{\text{selected by } \vdash}, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle}^{\text{selected by eval}}.\end{gathered}$$

This selection happens in two stages. The concrete states are determined by $\vdash$ as usual, using the fact that $\sigma_5(a) = \text{changed}$. The computation performed by the eval function is more involved. From Figure 3.17, we fix the dynamic run above as $\mathcal{D}$, fix a transition expression $\mathsf{next}(q, \mathsf{calls}(f))$ and then, based on the concrete state found by $\vdash$, we construct a map

$$\beta = [q \mapsto \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle].$$

Using the eval function definition in Figure 3.17, we apply the rule for the transition expression $T = \mathsf{next}(X, \mathsf{calls}(f))$ where X in our specification is the variable q that holds a concrete state. We must find a transition tr such that the following are true:

- If $\mathcal{D}, \mathsf{eval}(\mathcal{D}, \beta, X) \vdash \mathsf{calls}(f)$ then $\mathsf{time}(tr) > \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X))$, otherwise $\mathsf{time}(tr) \geq \mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X))$.
- $\mathcal{D}, tr \vdash \mathsf{calls}(f)$.
- There is no tr' with $\mathsf{time}(\mathsf{eval}(\mathcal{D}, \beta, X)) < \mathsf{time}(tr') < \mathsf{time}(tr)$ and $\mathcal{D}, tr' \vdash \mathsf{calls}(f)$.

Since $\mathcal{D}, \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle \nvdash \mathsf{calls}(f)$, we identify the transition $tr = \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle$. We have $\mathcal{D}, tr \vdash \mathsf{calls}(f)$ because $\sigma_6(f) = \text{called}$. Finally, there is no intermediate transition tr' because tr is the first transition found after the state at which we start. From this the selection can be seen.

3.5.4 A Total Semantics

With the concrete state and transition identified, in order to decide whether the dynamic run holds the property we consider, it remains only to measure the duration of the transition and decide whether it lies within $(0, 5)$. This process is generalised by the semantics for complete dynamic runs in Figures 3.18, 3.19 and 3.20. These figures cover a subset of the full syntax of CFTL (the cases not covered are a straightforward extension). Figure 3.18 gives the *base* of the semantics, that is, the rule for quantifiers, and the basic propositional connectives including $\wedge$ and $\vee$. Figure 3.19 gives the cases of the semantics for normal atoms. Figure 3.20 gives the cases of the semantics for mixed atoms. The semantics are divided across three figures to aid in a clean presentation. We will often simply refer to Figure 3.18 as the entire semantics, with the understanding that the other parts of the semantics are given in other figures.

$$
\begin{array}{lcl}
\mathcal{D} & \models & \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi \text{ iff} \\
 & & \quad \text{for all } \beta \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n) \text{ we have } \mathcal{D}, \beta \models \psi \\
\mathcal{D}, \beta & \models & \mathit{true} \\
\mathcal{D}, \beta & \models & \phi_1 \vee \phi_2 \text{ iff } \mathcal{D}, \beta \models \phi_1 \text{ or } \mathcal{D}, \beta \models \phi_2 \\
\mathcal{D}, \beta & \models & \phi_1 \wedge \phi_2 \text{ iff } \mathcal{D}, \beta \models \phi_1 \text{ and } \mathcal{D}, \beta \models \phi_2 \\
\mathcal{D}, \beta & \models & \neg\phi \text{ iff not } \mathcal{D}, \beta \models \phi
\end{array}
$$

Figure 3.18: The cases for quantification and propositional connectives for the $\models$ relation for the CFTL semantics.

In the semantics, we use the standard notation $\beta \dagger [q \mapsto c]$ to denote the map that agrees with β on all of its domain except for q, which it maps to c. We extend the comparison operator (generated by the *cmp* rule) to behave like a relation. We say that a complete dynamic run $\mathcal{D}$ *satisfies* a CFTL specification φ if and only if $\mathcal{D} \models \varphi$, and call this semantics the *total semantics* for CFTL.

3.5.5 An Example using the Total Semantics

Based on the total semantics, we now give an example of checking the complete dynamic run in Figure 3.2 with respect to the specification $\forall q \in \mathsf{changes}(a) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)$. We have already seen that the only concrete state in the dynamic run to be selected by the quantifier is $\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle$, so we only need to check the inner part of the specification at this concrete state. Hence, we are left to check whether

$$\mathcal{D}, [q \mapsto \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle] \models \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5).$$

The relevant transition, as identified previously, is

$$tr = \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle$$

so, by following the semantics, we have

$$\mathcal{D}, [q \mapsto \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle] \models \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5) \text{ iff } \mathsf{duration}(tr) \in (0, 5).$$

Finally, $\mathsf{duration}(tr) = 2.5 - 1.7 = 0.8 \in (0, 5)$, hence $\mathcal{D} \models \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)$.

3.5.6 Distinguishing between Normal and Weak Operators

The difference between the normal and weak versions of the next operators can be demonstrated by considering again the dynamic run in Figure 3.2 but with respect to the properties

$$\varphi_1 \equiv \forall q \in \mathsf{changes}(a) : \mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(g)))(x) = 10$$

$\mathcal{D}, \beta \models S(x) = v$ iff

$$\begin{cases} \mathsf{eval}(\mathcal{D}, \beta, S)(x) = v & \mathsf{eval}(\mathcal{D}, \beta, S) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \mathit{false} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null} \\ \mathit{true} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null}_W \end{cases}$$

$\mathcal{D}, \beta \models S(x) \in [n, m]$ iff

$$\begin{cases} \mathsf{eval}(\mathcal{D}, \beta, S)(x) \in [n, m] & \mathsf{eval}(\mathcal{D}, \beta, S) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \mathit{false} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null} \\ \mathit{true} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null}_W \end{cases}$$

$\mathcal{D}, \beta \models \mathsf{duration}(T) \in (n, m)$ iff

$$\begin{cases} \mathsf{duration}(\mathsf{eval}(\mathcal{D}, \beta, T)) \in (n, m) & \mathsf{eval}(\mathcal{D}, \beta, S) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \mathit{false} & \mathsf{eval}(\mathcal{D}, \beta, T) = \mathsf{null} \\ \mathit{true} & \mathsf{eval}(\mathcal{D}, \beta, T) = \mathsf{null}_W \end{cases}$$

Figure 3.19: The normal atom cases for the $\models$ relation for the CFTL semantics.

$$\varphi_2 \equiv \forall q \in \mathsf{changes}(a) : \mathsf{dest}_W(\mathsf{next}_W(q, \mathsf{calls}(g)))(x) = 10$$

which are distinguished by their use of the normal versus weak next and dest operators. In both cases, the $\vdash$ relation will identify $\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1]\rangle$ again as the single concrete state that satisfies the predicate $\mathsf{changes}(a)$, but the results of checking $\mathcal{D} \models \varphi_1$ and $\mathcal{D} \models \varphi_2$ will differ. To see why, we first look at the rule for $S(x) = v$ in total semantics in Figure 3.18:

$$\mathcal{D}, \beta \models S(x) = v \text{ iff } \begin{cases} \mathsf{eval}(\mathcal{D}, \beta, S)(x) = v & \mathsf{eval}(\mathcal{D}, \beta, S) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \mathit{false} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null} \\ \mathit{true} & \mathsf{eval}(\mathcal{D}, \beta, S) = \mathsf{null}_W \end{cases}$$

We need to compute $\mathsf{eval}(\mathcal{D}, \beta, S)$, where S in this case is $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(g)))$ and $\mathsf{dest}_W(\mathsf{next}_W(q, \mathsf{calls}(g)))$. From here, we follow the definition of eval for the dest operator:

$$\begin{array}{rcl} \mathsf{eval}(\mathcal{D}, \beta, \mathsf{dest}(T)) & = & \text{if } \mathsf{eval}(\mathcal{D}, \beta, T) \notin \{\mathsf{null}, \mathsf{null}_W\}, \text{ then } \mathsf{dest}(\mathsf{eval}(\mathcal{D}, \beta, T)) \\ & & \text{otherwise } \mathsf{null} \end{array}$$

$\mathcal{D}, \beta \models S_1(x_1)\ cmp\ S_2(x_2)$ iff

$$\begin{cases} \mathsf{eval}(\mathcal{D},\beta,S_1)(x_1) \\ \quad cmp\ \mathsf{eval}(\mathcal{D},\beta,S_2)(x_2) & \mathsf{eval}(\mathcal{D},\beta,S_1) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ & \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_2) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \\ \mathit{false} & \mathsf{eval}(\mathcal{D},\beta,S_1) = \mathsf{null} \\ & \quad \text{or } \mathsf{eval}(\mathcal{D},\beta,S_2) = \mathsf{null} \\ \\ \mathit{true} & \left(\begin{array}{l} \mathsf{eval}(\mathcal{D},\beta,S_1) = \mathsf{null}_W \\ \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_2) \neq \mathsf{null} \end{array}\right) \\ & \qquad \text{or} \\ & \left(\begin{array}{l} \mathsf{eval}(\mathcal{D},\beta,S_2) = \mathsf{null}_W \\ \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_1) \neq \mathsf{null} \end{array}\right) \end{cases}$$

$\mathcal{D}, \beta \models \mathsf{timeBetween}(S_1, S_2) \in (n, m)$ iff

$$\begin{cases} \mathsf{time}(\mathsf{eval}(\mathcal{D},\beta,S_2)) - \\ \quad \mathsf{time}(\mathsf{eval}(\mathcal{D},\beta,S_1)) \\ \quad\quad \in (n,m) & \mathsf{eval}(\mathcal{D},\beta,S_1) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ & \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_2) \notin \{\mathsf{null}, \mathsf{null}_W\} \\ \\ \mathit{false} & \mathsf{eval}(\mathcal{D},\beta,S_1) = \mathsf{null} \\ & \quad \text{or } \mathsf{eval}(\mathcal{D},\beta,S_2) = \mathsf{null} \\ \\ \mathit{true} & \left(\begin{array}{l} \mathsf{eval}(\mathcal{D},\beta,S_1) = \mathsf{null}_W \\ \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_2) \neq \mathsf{null} \end{array}\right) \\ & \qquad \text{or} \\ & \left(\begin{array}{l} \mathsf{eval}(\mathcal{D},\beta,S_2) = \mathsf{null}_W \\ \quad \text{and } \mathsf{eval}(\mathcal{D},\beta,S_1) \neq \mathsf{null} \end{array}\right) \end{cases}$$

Figure 3.20: The mixed atom cases for the $\models$ relation for the CFTL semantics.

The values of T we consider are $\mathsf{next}(q, \mathsf{calls}(g))$ and $\mathsf{next}_W(q, \mathsf{calls}(g))$, for which we need the following rule from eval:

$$\mathsf{eval}\,(\mathcal{D},\beta,\mathsf{next}(X,\mathsf{calls}(f))) = \begin{array}{l} \text{if } \mathsf{eval}(\mathcal{D},\beta,X) \notin \{\mathsf{null}, \mathsf{null}_W\} \text{ then} \\ \left(\begin{array}{l} tr \text{ if there is a } tr \text{ such that:} \\ \quad (\text{if } \mathcal{D}, \mathsf{eval}(\mathcal{D},\beta,X) \vdash \mathsf{calls}(f) \text{ then} \\ \quad\quad \mathsf{time}(tr) > \mathsf{time}(\mathsf{eval}(\mathcal{D},\beta,X)) \\ \quad \text{otherwise } \mathsf{time}(tr) \geq \mathsf{time}(\mathsf{eval}(\mathcal{D},\beta,X))) \\ \quad \text{and } \mathcal{D}, tr \vdash \mathsf{calls}(f) \\ \quad \text{and (there is no } tr' \text{ such that:} \\ \quad\quad\quad \mathsf{time}(\mathsf{eval}(\mathcal{D},\beta,X)) < \mathsf{time}(tr') < \mathsf{time}(tr) \\ \quad\quad\quad \text{and } \mathcal{D}, tr' \vdash \mathsf{calls}(f)) \\ \text{otherwise } \mathsf{null} \end{array}\right) \\ \text{otherwise } \mathsf{null} \end{array}$$

The value of X that we take here is q (the variable from the quantifier $\forall q \in \mathsf{changes}(a)$) which is not in $\{\mathsf{null}, \mathsf{null}_W\}$, so we must find some transition tr that satisfies the conditions given in the definition. In the dynamic run we consider, there is never any concrete state with a symbolic state σ such that $\sigma(g) = \text{changed}$ so the eval function gives null in the normal case and null_W in the weak case.

For the normal next, null is passed to dest, which also gives null. The semantics then gives *false* when given null from eval. For the weak case, null_W is passed to dest_W, which then gives null_W. Hence, in the weak case, the semantics gives *true*. This result highlights the motivation for having normal and weak operators in the total semantics: these essentially describe two policies of varying strictness. It should be reiterated at this point that, when we define a semantics over partial dynamic runs, this distinction between normal and weak operators will not exist. This is because failure to find a concrete state or transition in the future could just mean that we do not yet have enough information.

3.6 The Case with no Bindings

We have now defined the set of bindings that one can derive from a dynamic run with respect to a CFTL specification, along with the way in which the inner-most, quantifier-free part of a CFTL specification is evaluated given each of these bindings. Before concluding this chapter, we discuss the case in which no bindings are extracted from a dynamic run during monitoring.

Since the semantics introduced in Section 3.5 starts by evaluating the inner-most, quantifier free part of the specification with respect to each binding, one could say that, if there were no bindings, the specification would be satisfied (since there are no bindings to cause a violation).

However, in our use case setting it makes more sense to require that, to check whether a dynamic run satisfies a specification at all, there is at least one binding at which we can check the constraints it defines. Hence, rather than assuming that the semantics will give a true verdict for the case where no bindings are identified, from now on we will require that at least one binding be identified. Further, we will include this criterion in the list of criteria used to decide whether a CFTL specification is well-formed.

In order to formalise this criterion, we give two cases (one being the first quantifier in the sequence of quantifiers, and one being all others). Given a dynamic run $\mathcal{D}$, a CFTL specification with quantifiers $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$ (that is well-formed with respect to the original definition) is well-formed with respect to our new criterion if, for each Γ_i:

- If $i = 1$, there is some concrete state or transition, say s, in $\mathcal{D}$ such that $s \vdash \Gamma_i$.
- If $i > 1$, there is some concrete state or transition, say s, in $\mathcal{D}$ such that $s \vdash \Gamma_{i-1}$ and there is some concrete state or transition, say s', in $\mathcal{D}$ such that $s' \vdash \Gamma_i(s)$.

Formally, since our amended definition of well-formedness requires a dynamic run to check, we should say that a CFTL specification is well-formed *with respect to a dynamic run*. However, since this requirement is usually not violated in practice, we refer to a CFTL specification as simply being *well-formed*, regardless of a dynamic run.

3.7 Overview

In this chapter, we have introduced a new temporal logic for the specification of source code level properties over programs. This involved introducing a statically-computable representation of programs in the form of a directed graph which encodes reachability and state change information. From there, we introduced a new kind of trace to encode program runs. This allowed us to define the CFTL syntax and semantics, paving the way for the remainder of the contributions of this thesis.

Chapter 4

Monitoring and Instrumentation

This chapter will consider the problem of deciding whether a given run of a program satisfies a CFTL specification φ, building on the material presented in the papers [DR19b] and [DRF+19]. The approach that we present consists of both monitoring and instrumentation aspects.

Our first step is the introduction of a simple monitoring algorithm (Section 4.4) that processes a dynamic run and gives a true or false verdict to reflect whether or not the dynamic run is found to satisfy the given CFTL specification. We prove the correctness of this algorithm and derive an expression of its complexity. The complexity that we derive shows that this simple algorithm would not scale well given larger dynamic runs.

In order to remedy the situation, we develop an instrumentation approach (Section 4.7) that can 1) filter dynamic runs to include only the concrete states that are useful when monitoring for a CFTL specification; and 2) be used to reduce the time needed to find the relevant part of the monitoring state to update given a concrete state. We prove that removing from a dynamic run the concrete states that were not indicated as relevant by our instrumentation approach cannot result in a change in verdict given by monitoring. Finally, using our instrumentation approach, we give an optimised monitoring algorithm (Section 4.8).

4.1 The Intuition behind Monitoring

The monitoring problem for CFTL is that of taking a complete, most-general dynamic run $\mathcal{D}$ with a CFTL specification $\varphi \equiv \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \phi$ and deciding whether $\mathcal{D} \models \varphi$. The first attempt at solving this problem that we present boils down to two passes over each concrete state $s = \langle t, \sigma, \tau \rangle$ in a dynamic run. The first pass is to decide whether quantification should capture each concrete state. This means checking each concrete state s in the dynamic run against the criterion:

$s \vdash \Gamma_i$ or $\langle s', s \rangle \in \Gamma_i$
for s' the concrete state immediately before s in $\mathcal{D}$ and
$1 \leq i \leq n$.

The second pass is to decide whether a concrete state or transition contributes to the satisfaction of the quantifier-free part of the specification given a value for each of its variables from the

quantifiers. This means checking each concrete state s and transition $\langle s, s' \rangle$ in the dynamic run against the criterion:

$\mathsf{eval}(\mathcal{D}, \beta, X) = s$ or $\mathsf{eval}(\mathcal{D}, \beta, X) = \langle s', s \rangle$
for some s' in $\mathcal{D}$,
$X \in \{\mathsf{temp}(expr) : expr \in \mathsf{exprs}(\varphi)\} \cup \{\mathsf{temp}(\alpha) : \alpha \in A_\varphi \text{ and } \alpha \text{ is normal}\}$ and
binding β.

To illustrate why these two criteria are the starting point for a monitoring algorithm, we will demonstrate a naive algorithm by example on the sample dynamic run we have considered previously:

$$\langle 0, \sigma_0, [] \rangle, \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle,$$
$$\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle,$$
$$\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle, \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle,$$
$$\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle.$$

Figure 4.1 presents the satisfaction check for each concrete state/transition in this dynamic run with respect to the criteria given above, based on monitoring the CFTL specification

$$\forall q \in \mathsf{changes}(x) : q(x) < 5 \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5). \tag{4.1}$$

Hence, Figure 4.1 demonstrates the checks that one could perform given the dynamic run above, based on the two criteria. The criteria check at index 1 shows a concrete state being identified both as being relevant to a quantifier and as being relevant to an atom in the specification. At index 9, we see that a transition is found which is relevant to an atom in the specification. The monitoring algorithm that we develop will build on these checks.

We will now develop the machinery necessary for a general monitoring algorithm which will integrate these criteria. The first step in our development is the construction of machinery for remembering what we have observed once we determine that some concrete state or transition is needed to decide whether a CFTL specification holds.

4.2 Remembering Observed Information with Formula Trees

In Figure 4.1, we demonstrate that the criteria we have work; we use the $\vdash$ relation to determine whether a concrete state/transition should be selected by a quantifier, and then use eval to decide whether a part of the quantifier-free part of the specification can be given a value. The missing piece now is how to make information persistent for specifications such as (4.1), where information must be remembered between processing separate concrete states. For this specification, once we detect that $\langle 0.2, \sigma_1, [x \mapsto 3] \rangle \vdash \mathsf{changes}(x)$ and immediately decide whether $q(x) < 5$ using the same concrete state, we then have to remember that information for the later observation of a transition that corresponds to $\mathsf{next}(q, \mathsf{calls}(f))$. To do this, we opt for *formula trees* with additional structure to cope with CFTL's syntax. We remark that formula trees are essentially abstract syntax trees that are refined to fit the use case of monitoring for CFTL.

Index in $\mathcal{D}$	Concrete State	Criteria Check
0	$\langle 0, \sigma_0, [] \rangle$	-
1	$\langle 0.2, \sigma_1, [x \mapsto 3] \rangle$	$\langle 0.2, \sigma_1, [x \mapsto 3] \rangle \vdash \mathsf{changes}(x)$ and $\mathsf{eval}(\mathcal{D}, \beta, q) = \langle 0.2, \sigma_1, [x \mapsto 3] \rangle$ for $\beta = [q \mapsto \langle 0.2, \sigma_1, [x \mapsto 3] \rangle]$
2	$\langle 0.3, \sigma_2, [i \mapsto 0] \rangle$	-
3	$\langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle$	-
4	$\langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle$	-
5	$\langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle$	-
6	$\langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle$	-
7	$\langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle$	-
8	$\langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle$	-
9	$\langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle$	By checking indices 8 and 9: $\mathsf{eval}(\mathcal{D}, \beta, \mathsf{next}(q, \mathsf{calls}(f))) = \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle$ for $\beta = [q \mapsto \langle 0.2, \sigma_1, [x \mapsto 3] \rangle]$

Figure 4.1: Monitoring a dynamic run for a CFTL specification.

Definition 3 (Formula Tree)**.** *Given a quantifier-free CFTL specification ϕ (in* negated normal form*), the formula tree* $\mathsf{tree}(\phi)$ *of ϕ is a directed graph (N, E, n_s) where N is a set of nodes corresponding to subformulas of ϕ; E is a set of edges which either indicate the subformula relation (eg, $(\psi_1 \wedge \psi_2, \psi) \in E$ if $\psi_1 \wedge \psi_2$ is a subformula of ψ) or the sub-expression relation (eg, $(expr_1, expr_1\ cmp\ expr_2), (expr_2, expr_1\ cmp\ expr_2) \in E$ if $expr_1\ cmp\ expr_2$ is a mixed atom in ϕ); and n_s is the root corresponding to ϕ.*

For brevity, we denote a subtree by $\langle r, l_1, \ldots, l_k \rangle$ with root r and children $l_1, \ldots, l_k$ of r. Further, a singleton such as $\langle p \rangle$ or $\langle expr \rangle$ denotes a leaf, where p is a normal atom and $expr$ is an expression (see Section 3.4.1). The singletons $\langle \vee \rangle$ and $\langle \wedge \rangle$ denote the root vertices of subtrees representing disjunctions and conjunctions respectively. For the inner, quantifier-free part of a CFTL specification, we give a recursive construction of its formula tree via the function $\mathsf{tree}(\phi)$ using rules from the syntax in Figure 3.11. Recall that our well-formedness criterion requires that negation is propagated through to atoms.

$$
\begin{aligned}
\mathsf{tree}(p) &= \langle p \rangle \\
\mathsf{tree}(\neg p) &= \langle \neg p \rangle \\
\mathsf{tree}(expr) &= \langle expr \rangle \\
\mathsf{tree}(expr_1\ cmp\ expr_2) &= \langle cmp, \mathsf{tree}(expr_1), \mathsf{tree}(expr_2) \rangle \\
\mathsf{tree}(\textstyle\bigvee_{i=1}^{n} \phi_i) &= \langle \vee, \mathsf{tree}(\phi_1), \ldots, \mathsf{tree}(\phi_n) \rangle \\
\mathsf{tree}(\textstyle\bigwedge_{i=1}^{n} \phi_i) &= \langle \wedge, \mathsf{tree}(\phi_1), \ldots, \mathsf{tree}(\phi_n) \rangle
\end{aligned}
$$

Given a CFTL specification φ, the leaves of the formula tree $\mathsf{tree}(\phi)$ of its quantifier-free part ϕ correspond to 1) the atoms of φ, if the atoms are normal; and 2) the sub-expressions of the atoms of φ, if the atoms are mixed.

Constructing such a formula tree requires constructing a vertex for each *symbol* in the

quantifier-free part of the CFTL specification. The set of symbols $\mathsf{symbols}(\varphi)$ of a CFTL specification is the union of the set of atoms A_φ and the set of propositional connectives in $\{\vee, \wedge\}$. For example, the quantifier-free part

$$\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1 \vee \mathsf{dest}(\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))))(x) < 10$$

has symbols

$$\{\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1, \vee, \mathsf{dest}(\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))))(x) < 10\}.$$

Hence, the complexity of constructing such a tree is $O(|\mathsf{symbols}(\varphi)|)$, ie, linear in the size of the quantifier-free part of the specification.

4.2.1 Encoding Information held by Formula Trees

We now need a notion of how the information held by a formula tree is updated based on what is encountered when processing a dynamic run. Hence, we introduce the notion of an *interpretation* $\mathcal{I}$, which is a map from *atoms* and *expressions* to concrete states/transitions. Important to note is that the domain of an interpretation over the atoms/expressions in a CFTL specification may not include all that are found in the specification. This allows us to use interpretations to hold single concrete states or transitions extracted from a dynamic run.

Now, we consider formula trees under such interpretations. For example, the interpretation $\mathcal{I} = [q(x) \mapsto \langle t, \sigma, [x \mapsto 10]\rangle]$ (which maps from the expression $q(x)$ to a concrete state) would allow us to use the concrete state $\langle t, \sigma, [x \mapsto 10]\rangle$ to determine the value $q(x)$ by substituting q. Finally, for interpretations we use standard function modification notation $\mathcal{I} \dagger [v \mapsto s]$ to denote the interpretation that agrees with $\mathcal{I}$ everywhere apart from v, which it maps to s.

For a formula tree $\mathsf{tree}(\phi)$ and interpretation $\mathcal{I} = [v_i \mapsto s_i]$, we denote by $\mathsf{tree}(\phi) \downarrow \mathcal{I}$ the formula tree under the interpretation $\mathcal{I}$ and give a recursive definition for constructing it in Figure 4.2. In this definition, expressions such as $\mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I}$ *cmp* $\mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I}$ are always computable since these cases are only evaluated when both $\mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I}$ and $\mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I}$ have collapsed to values derived from a dynamic run. Further, given some *expr* taken from a mixed atom, we refer to the value to which it evaluates under the concrete state/transition given by some interpretation $\mathcal{I}$ as *expr under* $\mathcal{I}(\mathit{expr})$. Similarly, given some normal atom p and some concrete state/transition s, if the constraint expressed by p holds given s, then we write that $p =$ *true given* s. Conversely, if the constraint does not hold, then we write that $p =$ *false given* s. Since interpretations map atoms to concrete states/transitions, we can refer to the truth value of an atom *given the image of that atom under some interpretation* $\mathcal{I}$.

Ultimately, interpretations give us a way to 1) make persistent what we have processed so far while iterating over a dynamic run and, to improve efficiency, 2) avoid reprocessing all concrete states/transitions seen so far.

$$\mathsf{tree}(p) \downarrow \mathcal{I} = \begin{cases} \langle \mathit{true} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } p = \mathit{true} \text{ given } \mathcal{I}(p) \\ \langle \mathit{false} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } p = \mathit{false} \text{ given } \mathcal{I}(p) \\ \langle p \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{tree}(\neg p) \downarrow \mathcal{I} = \begin{cases} \langle \mathit{false} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } p = \mathit{true} \text{ given } \mathcal{I}(p) \\ \langle \mathit{true} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } p = \mathit{false} \text{ given } \mathcal{I}(p) \\ \langle p \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{tree}(\mathit{expr}) \downarrow \mathcal{I} = \begin{cases} \mathit{expr} \text{ under } \mathcal{I}(\mathit{expr}) & \mathit{expr} \in \mathsf{dom}(\mathcal{I}) \\ \langle \mathit{expr} \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{tree}(\mathit{expr}_1 \ \mathit{cmp} \ \mathit{expr}_2) \downarrow \mathcal{I} = \begin{cases} \langle \mathit{cmp}, \\ \quad \mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I}, \\ \quad \mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I} \rangle & \mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I} \text{ is a subtree} \\ & \quad \text{or } \mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I} \text{ is a subtree} \\ \langle \mathit{true} \rangle & \mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I} \ \mathit{cmp} \ \mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I} \\ \langle \mathit{false} \rangle & \neg(\mathsf{tree}(\mathit{expr}_1) \downarrow \mathcal{I} \ \mathit{cmp} \ \mathsf{tree}(\mathit{expr}_2) \downarrow \mathcal{I}) \end{cases}$$

$$\mathsf{tree}(\textstyle\bigvee_{i=1}^{n} \phi_i) \downarrow \mathcal{I} = \begin{cases} \langle \mathit{true} \rangle & \text{there is some } \phi_i \text{ such that:} \\ & \quad \mathsf{tree}(\phi_i) \downarrow \mathcal{I} = \langle \mathit{true} \rangle. \\ \langle \mathit{false} \rangle & \text{for every } \phi_i\text{:} \\ & \quad \mathsf{tree}(\phi_i) \downarrow \mathcal{I} = \langle \mathit{false} \rangle. \\ \langle \vee, \mathsf{tree}(\phi_1) \downarrow \mathcal{I}, & \text{otherwise} \\ \quad \ldots, \mathsf{tree}(\phi_n) \downarrow \mathcal{I} \rangle & \end{cases}$$

$$\mathsf{tree}(\textstyle\bigwedge_{i=1}^{n} \phi_i) \downarrow \mathcal{I} = \begin{cases} \langle \mathit{true} \rangle & \text{for every } \phi_i\text{:} \\ & \quad \mathsf{tree}(\phi_i) \downarrow \mathcal{I} = \langle \mathit{true} \rangle. \\ \langle \mathit{false} \rangle & \text{there is some } \phi_i \text{ such that:} \\ & \quad \mathsf{tree}(\phi_i) \downarrow \mathcal{I} = \langle \mathit{false} \rangle. \\ \langle \vee, \mathsf{tree}(\phi_1) \downarrow \mathcal{I}, & \text{otherwise} \\ \quad \ldots, \mathsf{tree}(\phi_n) \downarrow \mathcal{I} \rangle & \end{cases}$$

Figure 4.2: The construction of a formula tree under an interpretation.

4.3 Formula Trees with CFTL Specifications

Consider again the example $\forall q \in \mathsf{changes}(x) : q(x) < 5 \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)$. We first compute $\mathsf{tree}(\phi)$ for $\phi \equiv q(x) < 5 \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)$. We have

$$\begin{aligned} \mathsf{tree}(\varphi) &= \mathsf{tree}(q(x) < 5 \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)) \\ &= \mathsf{tree}(\neg(q(x) < 5) \vee \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)) \\ &= \langle \vee, \langle \neg(q(x) < 5) \rangle, \langle \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5) \rangle \rangle \end{aligned}$$

We now consider what an interpretation would look like in this case. The domain of such an interpretation is a (non-strict) subset of the set $\{\neg(q(x) < 5), \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5)\}$, hence one interpretation that could be constructed is

$$\mathcal{I} = [\neg(q(x) < 5) \mapsto \langle t, \sigma, [x \mapsto 4] \rangle]$$

which could be used to compute $\mathsf{tree}(\varphi) \downarrow \mathcal{I}$, which gives

$$\langle \vee, \langle \mathit{false} \rangle, \langle \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 5) \rangle \rangle.$$

One can then construct an extension of this interpretation $\mathcal{I} \dagger [\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f)) \in (0, 5) \mapsto tr]$ to give the formula tree $\langle \mathit{true} \rangle$ (we suppose that the observed transition tr satisfied the constraint placed by the specification, so the formula tree could be completely collapsed).

4.3.1 The Definitions of Formula Tree Update

We now introduce three functions to denote updates performed on formula trees in various ways. The functions are as such:

Update by Interpretation For a concrete state or transition s, we denote by

$$\mathsf{update}_{\mathcal{I}}(\mathsf{tree}(\varphi), [v \mapsto s])$$

the formula tree $\mathsf{tree}(\varphi)$ updated with respect to the interpretation $[v \mapsto s]$ sending the normal atom or expression v to the concrete state/transition s.

Update by Truth Value We denote by

$$\mathsf{update}_{\mathbb{B}}(\mathsf{tree}(\varphi), [p \mapsto b])$$

for $p \in A_\varphi$ and $b \in \mathbb{B}$ the formula tree $\mathsf{tree}(\varphi)$ updated by replacing all occurrences of the atom p with the truth value b and then collapsing as appropriate. We give a recursive definition in Figure 4.3.

Update by Binding Based on $\mathsf{update}_{\mathcal{I}}$, we denote by

$$\mathsf{update}_{\beta}(\mathsf{tree}(\varphi), \beta, s)$$

the version of the update function that takes a concrete state/transition along with a binding, rather than an interpretation. In this case, the update function will find all nodes in the formula tree containing either 1) an expression *expr* or 2) a normal atom p containing an expression *expr* for which $\mathsf{eval}(\mathcal{D}, \beta, expr) = s$. For each such node, the interpretation $[expr \mapsto s]$ (or $[p \mapsto s]$ in the case that a normal atom p was found to require s) will be constructed, allowing the process to be performed that is described in Figure 4.2 and represented by $\mathsf{update}_{\mathcal{I}}$.

The Differences between Update Approaches

The definition of formula tree update given by Figure 4.2 (and used by $\mathsf{update}_{\mathcal{I}}$) assumes an interpretation that sends *atoms* and *expressions* to concrete states or transitions.

As an example, for a normal atom p, the truth value of the atom is said to be true if the constraint expressed by the atom holds given the information contained in the concrete state or transition $\mathcal{I}(p)$. For example, if the interpretation $\mathcal{I}$ were defined as $[(q(x) < 10) \mapsto \langle t, \sigma, [x \mapsto 5]\rangle$ and a specification contained the atom $q(x) < 10$, then that atom could be replaced by $\langle true\rangle$ in the formula tree.

In the case of a mixed atom, for example $q(x) < q'(x)$, then one could construct the interpretation $\mathcal{I}$ with $[q(x) \mapsto 10]$ for $q(x)$ matching the *expr* case in the recursive definition. Updating the formula tree for the mixed atom would replace $q(x)$ with 10, but $q'(x)$ would stay so the formula tree would give no truth value.

The definition of formula tree update given by Figure 4.3 (and used by $\mathsf{update}_{\mathbb{B}}$) defines a formula tree under an interpretation that sends atoms (whether normal or mixed) immediately to truth values in $\mathbb{B}$. For example, given the atom $q(x) < 10$, then, rather than an interpretation $[(q(x) < 10) \mapsto s]$ for some concrete state s expected by the definition in Figure 4.2, an interpretation $\mathcal{I}$ of the form $[(q(x) < 10) \mapsto b]$ for $b \in \mathbb{B}$ would be expected. In this case, the language used in Figure 4.3 reflects the simpler situation: one can simply replace the relevant node in a formula tree with the image of the atom $q(x) < 10$ under $\mathcal{I}$.

We remark that these update functions and definitions are introduced to cater for various situations in the monitoring process. The complexity of the operations represented by the *update functions* is discussed in Section 4.6.

4.4 A Monitoring Algorithm

The first monitoring algorithm that we give, Algorithm 2, mirrors the total semantics: it takes a complete dynamic run $\mathcal{D}$ along with a CFTL specification φ and checks whether $\mathcal{D} \models \varphi$ by checking the satisfaction of the quantifier-free part of φ for each of the complete bindings computed by Algorithm 1.

We now describe the intuition behind the algorithm before proving its correctness. The general idea of the algorithm is to construct the set of complete bindings at line 2 and then initialise an empty map that will send these complete bindings to formula trees whose state will change as more concrete states are processed. The complexity of computing the set of bindings will be discussed in Section 4.6. The for-loop at line 5 handles formula tree initialisation.

$$\mathsf{tree}(p) \downarrow_* \mathcal{I} \;=\; \begin{cases} \langle \mathit{true} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(p) = \mathit{true} \\ \langle \mathit{false} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(p) = \mathit{false} \\ \langle p \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{tree}(\neg p) \downarrow_* \mathcal{I} \;=\; \begin{cases} \langle \mathit{false} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(p) = \mathit{true} \\ \langle \mathit{true} \rangle & p \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(p) = \mathit{false} \\ \langle p \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{tree}(\textstyle\bigvee_{i=1}^{n} \phi_i) \downarrow_* \mathcal{I} \;=\; \begin{cases} \langle \mathit{true} \rangle & \text{there is some } \phi_i \text{ such that :} \\ & \quad \phi_i \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(\phi_i) = \mathit{true} \\ \langle \mathit{false} \rangle & \text{for every } \phi_i \text{ such that } \phi_i \in \mathsf{dom}(\mathcal{I}) : \\ & \quad \mathcal{I}(\phi_i) = \mathit{false} \\ \langle \vee, \mathsf{tree}(\phi_1) \downarrow_* \mathcal{I}, & \text{otherwise} \\ \quad \ldots, \mathsf{tree}(\phi_n) \downarrow_* \mathcal{I} \rangle & \end{cases}$$

$$\mathsf{tree}(\textstyle\bigwedge_{i=1}^{n} \phi_i) \downarrow_* \mathcal{I} \;=\; \begin{cases} \langle \mathit{true} \rangle & \text{for every } \phi_i \text{ such that } \phi_i \in \mathsf{dom}(\mathcal{I}) : \\ & \quad \mathcal{I}(\phi_i) = \mathit{true} \\ \langle \mathit{false} \rangle & \text{there is some } \phi_i \text{ such that :} \\ & \quad \phi_i \in \mathsf{dom}(\mathcal{I}) \text{ and } \mathcal{I}(\phi_i) = \mathit{false} \\ \langle \wedge, \mathsf{tree}(\phi_1) \downarrow_* \mathcal{I}, & \text{otherwise} \\ \quad \ldots, \mathsf{tree}(\phi_n) \downarrow_* \mathcal{I} \rangle & \end{cases}$$

Figure 4.3: An alternative construction of a formula tree under an interpretation mapping atoms to truth values.

Algorithm 2 Check the dynamic run $\mathcal{D}$ for satisfaction of the CFTL specification $\forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n : \phi$.

```
1:  %Construct a set of complete bindings based on D and the quantifiers we have
2:  B ← bindings_φ(D, ∀1 s1 ∈ Γ1 : ... : ∀n sn ∈ Γn)
3:  C ← []                          ▷ Initialise an empty map from complete bindings to formula trees
4:  %Initialise the map C with formula trees
5:  for β in B do
6:    C ← C † [β ↦ tree(φ)]
7:  prev ← ⟨0, [], []⟩                                  ▷ Assign an empty concrete state
8:  %Iterate through D, updating the formula trees to obtain verdicts
9:  for curr ∈ D do
10:   for (β, T) in C do
11:     T' ← update_β(T, β, ⟨prev, curr⟩)
12:     if T' = false then return false
13:     C ← C † (β ↦ T')
14:     T' ← update_β(T, β, curr)
15:     if T' = false then return false
16:     C ← C † (β ↦ T')
17:   prev ← curr
18: %Find the complete bindings whose formula trees have not reached a verdict.
19: %For each such binding, we find the atoms for which a truth value has not been determined
20: %and decide what the truth value must be based on whether the temporal operator involved
21: %is normal or weak.
22: for (β, T) in C do
23:   if T ≠ true and resolve(T) = false then
24:     return false
25: return true
```

Line 8 initialises the previous concrete state variable. This is used to attempt to update formula trees with both transitions (pairs of concrete states) and individual concrete states.

The for-loop at line 9 iterates through the dynamic run and, for each concrete state, iterates through all formula trees. For each formula tree, an update is attempted using either the current concrete state, or the transition leading to it. For both updates, we check for the formula tree collapsing to *false* (in which case the algorithm terminates with *false*) and a value not equal to *true or false*.

The for-loop at line 22 is responsible for the case in which there was not enough information in the dynamic run to collapse all formula trees associated with complete bindings to a verdict. In this case, we run the resolve operation on each relevant formula tree. This function, given in Algorithm 3, determines all parts of a formula tree for which information is still needed and then automatically resolves them based on whether normal or weak temporal operators are used. This mirrors the behaviour described in the total semantics. An alternative kind of interpretation is also required, which is defined in Figure 4.3. Next, we discuss correctness (in Section 4.5) and complexity (in Section 4.6).

Algorithm 3 The resolve function: Given a formula tree T which has not reached a truth value, replace atoms for which there is not yet a truth value with truth values based on weak vs normal temporal operators.

```
1: %Determine all expressions and atoms that have not been given a value in T.
2: atoms ← {ψ : ψ is an atom and has no truth value in T}
3: %The order in which we iterate over atoms does not matter,
4: %so here we implicitly choose an arbitrary ordering.
5: for ψ ∈ atoms do
6:   if ψ is normal then
7:     op ← temp(ψ)                          ▷ Get the final temporal operator used in ψ
8:     if op is weak then
9:       T ← update_𝔹(T, β, [ψ ↦ true])
10:    else
11:      T ← update_𝔹(T, β, [ψ ↦ false])
12:  else
13:    %We have a mixed atom - there are more cases to check.
14:    lhsFinal ← temp(lhs(ψ))
15:    rhsFinal ← temp(rhs(ψ))
16:    if neither lhs(ψ) nor rhs(ψ) has a value in T then
17:      if at least one of lhsFinal and rhsFinal is normal then
18:        T ← update_𝔹(T, β, [ψ ↦ false])
19:      else
20:        T ← update_𝔹(T, β, [ψ ↦ true])
21:    else if lhs(ψ) does not have a value in T then
22:      if lhsFinal is weak then
23:        T ← update_𝔹(T, β, [ψ ↦ true])
24:      else
25:        T ← update_𝔹(T, β, [ψ ↦ false])
26:    else if rhs(ψ) does not have a value in T then
27:      if rhsFinal is weak then
28:        T ← update_𝔹(T, β, [ψ ↦ true])
29:      else
30:        T ← update_𝔹(T, β, [ψ ↦ false])
31: %By this point, T is guaranteed to be either true or false.
32: return T
```

4.5 Correctness of the Naive Algorithm

We now prove that Algorithm 2 is correct, that is, for a dynamic run $\mathcal{D}$ and CFTL property $\varphi \equiv \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi$, we have $\mathcal{D} \models \varphi$ if and only if Algorithm 2 gives *true*. We argue this in multiple parts:

1. We show that the set of bindings considered by the semantics is precisely the set of complete bindings generated by Algorithm 2.

   This is important because the existence of a complete binding in C provides the opportunity for failure. If the set of complete bindings generated misses some that are considered by the semantics, a failure may be missed. Conversely, if the set of complete bindings contains a binding not considered by the semantics (hence, an incorrect binding), failure could be generated unnecessarily (and incorrectly).

2. We show that the evaluation that takes place for each binding based on concrete states/-transitions found in the dynamic run matches the evaluation in the semantics.

   This is important because the semantics has access to any concrete state or transition that is needed in the future, but Algorithm 2 must perform an incremental update. We show that this incremental update has the same effect as the instantaneous update performed in the semantics.

For our first lemma concerning whether the total semantics and the algorithm generate the same set of bindings, the first monitoring algorithm that we have given only evaluates the quantifier-free part of the specification for complete bindings. This behaviour is matched by the total semantics

Lemma 1. *For a dynamic run $\mathcal{D}$ and CFTL specification $\varphi = \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \phi$, the set of bindings considered by the semantics in Figure 3.18 is precisely the set $\mathsf{dom}(C)$ consisting of complete bindings at the end of a run of Algorithm 2.*

Proof. The algorithm for computation of bindings, Algorithm 1, is correct (by Theorem 1). Hence, the bindings used in the semantics and Algorithm 2 are the same. □

Our next lemmata are necessary because the evaluation of the quantifier-free part of the specification performed by the semantics and then by the algorithm is done differently. The semantics assumes that any concrete states/transitions in the future are available for evaluation, hence as soon as a complete binding is formed, the quantifier-free part of the specification can be instantly evaluated at this binding. The algorithm differs from this because it must wait until it has reached those concrete states/transitions in the future in order to evaluate the formula tree for a given concrete binding.

We first give a lemma regarding the commutativity of formula tree updates under the circumstances observed when considering dynamic runs. If such updates are commutative, then we can merge the updates into one update with a single interpretation (whose domain has no guaranteed total ordering, so commutativity is required).

Lemma 2. *Formula tree updates using $\downarrow$ are commutative when considering interpretations derived from dynamic runs with respect to CFTL specifications.*

Proof. Given a formula tree $\mathsf{tree}(\varphi)$, consider the interpretations $\mathcal{I}_1, \mathcal{I}_2$. We want to show that $(\mathsf{tree}(\varphi) \downarrow \mathcal{I}_1) \downarrow \mathcal{I}_2 = (\mathsf{tree}(\varphi) \downarrow \mathcal{I}_2) \downarrow \mathcal{I}_1$. We will show that the only time that this is not the case cannot happen when monitoring dynamic runs with respect to CFTL specifications.

Suppose that $\mathcal{I}_1 = [\psi \mapsto s]$ and $\mathcal{I}_2 = [\psi \mapsto s']$ such that ψ is a normal atom that becomes *true* when s is used to evaluate it and *false* when s' is used. Using this configuration, commutativity may break because, if $\mathcal{I}_1$ were used first the formula tree may collapse to a different verdict to if $\mathcal{I}_2$ were used first. However, such a situation cannot happen because all temporal operators in CFTL identify a unique concrete state or transition (ie, the concrete state or transition required for each atom or expression is observed at most once). From this, since the conjunction and disjunction operators encoded by formula trees are commutative, formula tree updates with $\downarrow$ are commutative. □

We now explain how this commutativity result means that updates can be merged, meaning incremental versus instantaneous formula tree updates have the same result.

Lemma 3. *For a formula tree* $\mathsf{tree}(\varphi)$ *and family of interpretations* $\mathcal{I}_1, \ldots, \mathcal{I}_2$ *derived from a dynamic run* $\mathcal{D}$ *based on* φ *with* $|\mathsf{dom}(\mathcal{I}_1)| = \cdots = |\mathsf{dom}(\mathcal{I}_2)| = 1$*,*

$$(\ldots(\mathsf{tree}(\varphi) \downarrow \mathcal{I}_1)\ldots) \downarrow \mathcal{I}_n) = \mathsf{tree}(\varphi) \downarrow (\mathcal{I}_1 \dagger \ \ldots \ \dagger \mathcal{I}_n).$$

Proof. Since, by Lemma 2, $\downarrow$ is commutative when taking concrete states and transitions from a dynamic run based on a CFTL specification, the order in which the updates are performed does not matter. Given that the recursive definition of $\downarrow$ requires no order to be placed on the domain of the interpretation, the result follows. □

From this, we can conclude that the instantaneous evaluation performed by the semantics is equivalent to the incremental update of formula trees performed by the monitoring algorithm, since incremental update can be expressed as a single update in which the original order of updates does not matter.

Theorem 2. *For a dynamic run* $\mathcal{D}$ *and CFTL specification* $\varphi \equiv \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi$*,* $\mathcal{D} \models \varphi$ *if and only if Algorithm 2 gives true.*

Proof. By Lemma 1, the correct bindings are generated by the algorithm and, by Lemma 3, the formula tree evaluation has the same result as the evaluation performed in the semantics.

The only thing remaining to show is that weak and normal operators are evaluated in the same way between the semantics and the algorithm. The algorithm takes into account these two types of operators by processing all non-collapsed formula trees at the end, once the dynamic run has been processed. This is equivalent to the forward search performed by the semantics; in both cases, the whole future of the dynamic run is visible. Further, the conditions in the resolve procedure (Algorithm 3) are identical to the conditions checked in the semantics. □

4.6 Complexity of the Naive Algorithm

We derive the complexity for our initial monitoring algorithm when monitoring a dynamic run $\mathcal{D}$ with respect to a CFTL specification φ with n quantifiers.

4.6.1 Binding Computation

The starting point to derive the complexity of our initial monitoring algorithm is to observe that the algorithm for $\mathsf{bindings}_\varphi$ must iterate through the entire dynamic run, which we will say has length $|\mathcal{D}|$. Inside the main loop of the algorithm is another loop which iterates through existing partial bindings to determine whether they can be extended using the current concrete state/transition.

We need an upper bound on the number of partial and complete bindings that can be generated while checking a dynamic run with respect to a specification with n quantifiers (quantifiers influence the number of bindings, no other part of the specification does). $|\mathcal{D}|^n$ is

far too loose since each quantifier in a CFTL specification depends on the previous, meaning many of the $|\mathcal{D}|^n$ would never be generated. To determine a tighter bound, we first observe that the states of a dynamic run can be labelled with integers based on the total order induced on them by their timestamps. Hence, the *earliest* concrete state has label 0, the next has label 1, and so on. Now, we fix a binding $[q_1 \mapsto s_1, \ldots, q_n \mapsto s_n]$ and notice that $\mathsf{time}(s_1) \leq \mathsf{time}(s_2) \leq \cdots \leq \mathsf{time}(s_n)$. Finally, instead of considering actual concrete states, we instead consider their labels which, for a CFTL specification with n quantifiers, lie inside the hypercube

$$\{(x_1, \ldots, x_n) \in \mathbb{R}^n : 0 \leq x_i \leq n\}.$$

It follows that the total number of bindings is the total volume of this hypercube in which $x_1 \leq \cdots \leq x_n$. This is given by the volume integral which encodes $x_1 \leq \cdots \leq x_n$ in the limits along each dimension:

$$\begin{aligned}
\int_{x_n=0}^{|\mathcal{D}|} \left(dx_n \int_{x_{n-1}=0}^{x_n} \left(dx_{n-1} \cdots \int_{x_1=0}^{x_2} dx_1 \right) \right) &= \int_{x_n=0}^{|\mathcal{D}|} \left(dx_n \int_{x_{n-1}=0}^{x_n} \left(dx_{n-1} \cdots \int_{x_2=0}^{x_3} x_2 \; dx_2 \right) \right) \\
&= \int_{x_n=0}^{|\mathcal{D}|} \left(dx_n \int_{x_{n-1}=0}^{x_n} \left(dx_{n-1} \cdots \int_{x_3=0}^{x_4} \frac{x_3^2}{2} \; dx_3 \right) \right) \\
&= \frac{|\mathcal{D}|^n}{n!}
\end{aligned}$$

hence the number of bindings given a dynamic run $\mathcal{D}$ and a CFTL specification with n quantifiers is $|\mathcal{D}|^n/n!$, meaning the number of partial and complete bindings is $\Sigma_{p=1}^{n}(|\mathcal{D}|^p/p!)$. It follows that the complexity of binding derivation is $O(|\mathcal{D}|^{n+1}/n!)$. Notice that, for $n = 2$ quantifiers, this gives $|\mathcal{D}|^2/2$, which means we have to iterate over $\mathcal{D}$ to generate, for each concrete state, at most one binding. Then, for each of the at most $|\mathcal{D}|$ bindings, we must iterate through the dynamic run again to find possible extensions. The factor of $1/2$ indicates that we will take at most half of the total number of pairs one can form of concrete states/transitions from $\mathcal{D}$ (a constraint resulting from the requirement on concrete states' timestamps). Further, this assumption that $n = 2$ is reasonable since this is the most quantifiers we ever need in real use cases (described in Chapter 7). Ultimately, this double-iteration is of course a naive solution; we introduce significant optimisations later.

4.6.2 Evaluating the Quantifier-free part of a Specification

Evaluating the formula tree associated with each of the $|\mathcal{D}|^n/n!$ complete bindings generated requires iteration over the dynamic run, then iteration over the bindings, then two formula tree updates. Hence we have $C_{\mathsf{tree}}|\mathcal{D}|^{n+1}/n!$ where C_{tree} is the complexity of the formula tree updates that we will now determine.

A single call to $\mathsf{update}_\beta(T, \beta, s)$ for a concrete state s requires that the expression or atom is found for which $\mathsf{eval}(\mathcal{D}, \beta, \psi) = s$, and then that interpretations are constructed. The iteration through the atoms/expressions is worst case $2|A_\varphi| \in O(|A_\varphi|)$ for the set A_φ of atoms, if every atom were mixed (then there would be $2|A_\varphi|$ expressions), and the identification of a concrete state/transition for $\mathsf{eval}(\mathcal{D}, \beta, \psi)$ is worst case $|\mathcal{D}|$ (since at worst we have to check

every concrete state in the dynamic run). The update of the formula tree given a concrete state/transition via an interpretation then has worst case complexity m^h for m the maximum arity of any boolean operator in the specification, and h the height of the formula tree, since all nodes must be checked. So, the worst case complexity of the evaluation of all formula trees is $O(|A_\varphi| m^h |\mathcal{D}|^{n+2}/n!)$. In practice, m^h and $|A_\varphi|$ are usually very small in relation to the length of a dynamic run, so we can count these as negligible. For example, in the applications of this material described in Chapter 7, $|A_\varphi|$ never usually goes above 2 (but we expect specifications to increase slightly in complexity during future use cases) and both m and h are usually no more than 2.

4.6.3 Resolving Formula Trees

Resolving a formula tree again requires at most m^h steps to check every node, and there can be at most as many formula trees to resolve as there are complete bindings, hence $|\mathcal{D}|^n/n!$. Given that, once again we can consider the resolution procedure to take a negligible amount of time in practice, we simply count the number of complete bindings again.

4.6.4 The Final Complexity

We have $O(|\mathcal{D}|^n/n!)$ for binding computation, $O(|\mathcal{D}|^{n+2}/n!)$ for formula tree updates and $O(|\mathcal{D}|^n/n!)$ for formula tree resolution, hence

$$O(|\mathcal{D}|^n/n! + |\mathcal{D}|^{n+2}/n! + |\mathcal{D}|^n/n!) = O\left(\frac{|\mathcal{D}|^n}{n!}\left(1 + |\mathcal{D}|^2 + 1\right)\right).$$

Since n is the number of quantifiers in the CFTL specification being checked and the length of the dynamic run can grow independently (well beyond the number of quantifiers), we can simplify this to

$$O(|\mathcal{D}|^{n+2}).$$

We observe that n is often far smaller than $|\mathcal{D}|$ and, as constructs in programs such as loops iterate more times based on a program parameter change, n depends only on the specification. Further, the relevant concrete states/transitions in a dynamic run are often far fewer than the irrelevant ones, so this initial monitoring algorithm does a lot of unnecessary work (for example, with formula tree update, which checks whether every concrete state/transition processed is relevant). In the remainder of this chapter, we show how to improve the situation with instrumentation for CFTL specifications.

4.7 Instrumentation to Filter and Organise Dynamic Runs

Algorithm 2 iterates through a dynamic run which is assumed to have enough information, hence we normally consider it in the context of a most-general dynamic run (recall that this means that every transition corresponds to a single edge in the symbolic control-flow graph). Most-general dynamic runs have two main problems:

1. In practice, they represent recording *everything* from the program under scrutiny. This is infeasible because it would lead to extremely high overhead.

2. They result in a lot of work done by the monitoring algorithm to decide whether each concrete state and transition is actually needed.

We will now develop a method for safely removing unnecessary concrete states from a dynamic run. We say *safely* because we must not throw away any concrete states that are required to check our CFTL specification, since this would result in incorrect results from monitoring. Conversely, we do not want a criterion for throwing away concrete states that is too weak; we aim to reduce the work the monitoring algorithm performs as much as possible.

Our approach will also give a way to index formula trees so that lookup becomes much faster since, in Algorithm 2, bindings must be extended by searching through potentially many bindings blindly (which makes for a lot of wasted work in most cases, where the concrete states/transitions that contribute to bindings are a small percentage of the full set found in a dynamic run).

4.7.1 Defining a Strategy

We begin by making the observation that, in the total semantics given in Figure 3.18, the criteria used to decide whether a concrete state or transition should be identified by the $\vdash$ relation uses only information from the symbolic control-flow graph. Take, for example, the condition for $\mathcal{D}, \langle t, \sigma, \tau \rangle \vdash \mathsf{changes}(x)$:

$$\mathcal{D}, \langle t, \sigma, \tau \rangle \vdash \mathsf{changes}(x) \text{ iff } \sigma(x) = \text{changed}.$$

This means that we can statically identify symbolic states that may generate concrete states and transitions identified by quantifiers. Further, given that the eval function also uses only information from the symbolic control-flow graph, we can do the same there. This makes sense since dynamic runs can be seen as paths through symbolic control-flow graphs.

We now exploit this relationship to construct a set of symbolic states in the symbolic control-flow graph of a program. Each symbolic state in this set will be such that only the concrete states containing that symbolic state will be needed for 1) bindings or 2) evaluation of truth values of atoms. We will construct a set Inst of *instrumentation points*, that is, symbolic states with which the concrete states relevant to our specification will be associated. Our approach will consist of two steps:

1. *Inspecting the quantifiers* to determine the sets of symbolic states/edges that will be held by concrete states/transitions that will contribute to the bindings derived from a dynamic run.

2. Based on the result of inspecting quantifiers, *inspecting the atoms* in the specification to determine the symbolic states/edges that will be held by certain concrete states/transitions. The concrete states/transitions with which we will concern ourselves will be the ones that contribute to the checking of constraints expressed by atoms.

$$\begin{aligned}
\sigma &\in \mathsf{support}_\forall(\mathsf{changes}(x)) \text{ if } \sigma(x) = \text{changed} \\
\langle \sigma_1, \sigma_2 \rangle &\in \mathsf{support}_\forall(\mathsf{calls}(f)) \quad \text{if } \langle \sigma_1, \sigma_2 \rangle \in E \text{ and } \sigma_2(f) = \text{called}
\end{aligned}$$

With Γ_i the predicate of the previous quantifier:

$$\begin{aligned}
\sigma \in \mathsf{support}_\forall(\mathsf{future}(q_i, \mathsf{changes}(x)) \text{ if} \\
\quad \exists \sigma' \in \mathsf{support}_\forall(\Gamma_i) : \\
\quad\quad \mathsf{reaches}(\sigma', \sigma) \text{ and } \sigma(x) = \text{changed} \\
\langle \sigma_1, \sigma_2 \rangle \in \mathsf{support}_\forall(\mathsf{future}(q_i, \mathsf{calls}(f))) \text{ if} \\
\quad \exists \sigma' \in \mathsf{support}_\forall(\Gamma_i) : \\
\quad\quad \mathsf{reaches}(\sigma', \sigma_1) \text{ and } \langle \sigma_1, \sigma_2 \rangle \in E \text{ and } \sigma_2(f) = \text{called}
\end{aligned}$$

Figure 4.4: The symbolic support of predicates used by quantifiers.

4.7.2 Inspecting Quantifiers

The bindings derived from a dynamic run with respect to a CFTL specification can consist of a mixture of concrete states and transitions. These concrete states and transitions contain symbolic states. We now present our approach for inspecting the quantifiers of a CFTL specification in order to determine such symbolic states.

Symbolic Support for Quantifiers

Our first step is to introduce the *symbolic support* of the predicates on which quantifiers are based, along with atoms and expressions in the quantifier-free part of the specification. For this, we fix the symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ of a program P. For the specification $\varphi \equiv \forall_1 q_1 \in \Gamma_1 : \ldots : \forall_n q_n \in \Gamma_n : \phi$, we denote the symbolic support of a predicate Γ_i by $\mathsf{support}_\forall(\Gamma_i)$. Figure 4.4 gives the definition of the symbolic support for predicates used in quantifiers. Further, $\mathsf{reaches}(\sigma_1, \sigma_2)$ is true if it is possible to reach σ_2 from σ_1 in the symbolic control-flow graph (reachability is a transitive closure of E). Reachability, $\mathsf{reaches}(\sigma_1, \sigma_2)$, on a symbolic control-flow graph is decidable because such a graph is finite and statically computable from a program constructed using the grammar we assume in Figure 3.1 (page 55).

Hence, for a predicate Γ_i taken from the sequence of quantifiers in a CFTL specification, $\mathsf{support}(\Gamma_i)$ denotes the set of symbolic states that could appear in the concrete states identified by the predicate Γ_i in a dynamic run. Notice that symbolic support only identifies symbolic states; if Γ_i identifies concrete states in the dynamic run, then the correspondence is straightforward. If Γ_i identifies transitions, then the symbolic support will contain the symbolic states that correspond to the concrete states at either side of the transitions.

To cater for CFTL specifications having multiple quantifiers, we now introduce a way to collect together the symbolic supports from each quantifier. We use a structure called a *static binding*, named as such to highlight the fact that each binding generated during monitoring of a dynamic run corresponds to one of these static bindings. We define them precisely in Definition 4.

Definition 4 (Static Binding)**.** *A* static binding *derived from a symbolic control-flow graph* $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ *with respect to a CFTL specification* $\varphi \equiv \forall_1 q_1 \in \Gamma_1 : \ldots : \forall_n q_n \in \Gamma_n : \phi$

Algorithm 4 Derive the set of static bindings from the symbolic control-flow graph $\langle V, E, v_s \rangle$ with respect to the CFTL specification $\varphi \equiv \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi$

1: $\mathcal{B}_\varphi \leftarrow \{\}$ ▷ Initialise the final set of static bindings
2: $\mathsf{allBindings} \leftarrow \{\}$ ▷ Initialise the intermediate set of static bindings
3: **for** (q_i, Γ_i) from φ in order from $i = 1, \ldots, n$ **do**
4: **for** $v \in V \cup E$ **do**
5: **if** $v \in \mathsf{support}_\forall(\Gamma_i)$ **then**
6: *%For every static binding that v can extend, perform the extension*
7: $\mathsf{allBindings} \leftarrow \mathsf{allBindings} \cup \{\theta \dagger [q_i \mapsto v] : \theta \in \mathsf{allBindings} \wedge (i > 1 \rightarrow \mathsf{reaches}(\theta(q_{i-1}), v))\}$
8: *%Keep only the static bindings whose domain includes every variable from the specification*
9: **for** $\theta \in \mathsf{allBindings}$ **do**
10: **if** $|\mathsf{dom}(\theta)| = n$ **then**
11: $\mathcal{B}_\varphi \leftarrow \mathcal{B}_\varphi \cup \{\theta\}$

is a map $\mathsf{vars}(\varphi) \to V \cup E$*, ie, a map sending variables in* φ *to symbolic states in* V *or pairs of symbolic states in* E*. We denote a static binding by* θ*.*

As an example, let θ be a static binding derived from a symbolic control-flow graph with respect to the CFTL specification $\forall q \in \mathsf{changes}(x) : \forall t \in \mathsf{future}(q, \mathsf{calls}(f))$. Then we could have $\theta = [q \mapsto \sigma_1, t \mapsto \langle \sigma_2, \sigma_3 \rangle]$, where necessarily $\sigma_1(x) = \text{changed}$ and $\sigma_3(f) = \text{called}$.

The motivation for static bindings is to say that, if we have quantifiers $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$, then a symbolic state or pair identified for some Γ_i with $i > 1$ should be in the same static binding as the symbolic state or pair from the symbolic support of Γ_{i-1} that was used by the rule in Figure 4.4 to determine it. This intuition for static bindings will help us later to organise formula trees and make their lookup more efficient. Algorithm 4 gives a straightforward algorithm for the derivation of the set $\mathcal{B}_\varphi$ of static bindings given a symbolic control-flow graph (implicitly) and a CFTL specification. We now give a correctness argument.

Theorem 3. *Fix a dynamic run* $\mathcal{D}$ *and a CFTL specification* φ*. For every* $\beta \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ *derived from* $\mathcal{D}$*, the set* $\mathcal{B}_\varphi$ *computed by Algorithm 4 contains a static binding* θ *such that for every* $q_i, q_{i+1} \in \mathsf{vars}(\varphi)$ *with* $(1 \leq i < n)$*,* $\theta(q_{i+1})$ *is reachable in the symbolic control-flow graph from* $\theta(q_i)$ *and, for every* $q_i \in \mathsf{vars}(\varphi)$*, we have:*

- *If* $\beta(q_i)$ *is a concrete state* $s = \langle t, \sigma, \tau \rangle$ *such that* $s \vdash \Gamma_i$*, then* $\theta(q_i) = \sigma \in \mathsf{support}_\forall(\Gamma_i)$*.*
- *If* $\beta(q_i)$ *is a transition* $tr = \langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle$ *such that* $tr \vdash \Gamma_i$*, then* $\theta(q_i) = \langle \sigma, \sigma' \rangle \in \mathsf{support}_\forall(\Gamma_i)$*.*

The intuition of this theorem is as follows: the first criterion placed on static bindings ensures that, if a q' is after q in the sequence of quantifiers, then not only will it hold a concrete state/transition that occurs after q in the dynamic run, but this will translate into $\theta(q')$ being reachable from $\theta(q)$ in the symbolic control-flow graph. Hence, the first criterion ensures that instrumentation yields static bindings that correspond to bindings derived from dynamic runs. The second criteria ensure that the images of variables under static bindings actually correspond to the images of variables under bindings in $\mathsf{bindings}_\varphi(\mathcal{D}, \forall q_1 \in \Gamma_1 : \cdots : q_n \in \Gamma_n)$.

Proof. Fix a binding $\beta = [q_1 \mapsto s_1, \ldots, q_n \mapsto s_n]$ derived from a dynamic run $\mathcal{D}$ with respect to a CFTL specification φ. Take the quantifier $\forall q_1 \in \Gamma_1$. The algorithm would iterate through all $v \in V \cup E$ to find those v with $v \in \mathsf{support}_\forall(\Gamma_1)$. For each of these, a new binding θ would be generated (for q_1, the empty map will be extended). Hence, this step will result in the set of bindings $\{[q_1 \mapsto v] : v \in \mathsf{support}_\forall(\Gamma_1)\}$. Similarly to the binding construction performed during monitoring, progressing by extending existing bindings, by definition of $v \in \mathsf{support}_\forall(\Gamma_i)$, the second criterion is satisfied because bindings are extended for each Γ_i processed.

For the first criterion, we observed that the definition of the symbolic support of a predicate Γ constructs a set of symbolic states or edges based on whether there is *some* symbolic state or edge from which a new addition is reachable. This alone would not guarantee the first condition, since we could generate a static binding such that $\theta(q_{i+1})$ is in fact reachable from $\theta'(q_i)$ for some other static binding θ'. The algorithm checks a stronger condition; that the new v to be added should be reachable from $\theta(q_i)$. Hence, the first criterion that $\theta(q_{i+1})$ should be reachable from $\theta(q_i)$ is satisfied. □

With this result, we have that Algorithm 4 necessarily generates *only* the static bindings that could be associated with bindings generated while monitoring a dynamic run. As a corollary, we have the important result that each binding generated by Algorithm 2 corresponds to a single static binding.

Corollary 1. *For a dynamic run $\mathcal{D}$ and CFTL specification φ, for every $\beta \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, there is a $\theta \in \mathcal{B}_\varphi$ such that, for every $q \in \mathsf{dom}(\beta)$, if $\beta(q)$ is a concrete state $\langle t, \sigma, \tau\rangle$ then $\theta(q) = \sigma$ and if $\beta(q)$ is a transition $\langle t, \sigma, \tau\rangle, \langle t', \sigma', \tau'\rangle$ then $\theta(q) = \langle \sigma, \sigma'\rangle$.*

Given a set $\mathcal{B}_\varphi$ of static bindings, we give an example to motivate our next step. Consider, from our initial introduction to symbolic control-flow graphs, the graph in Figure 4.5 along with the CFTL specification $\forall q \in \mathsf{changes}(y) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$. In this case, we compute the symbolic support of $\mathsf{changes}(y)$ to determine a set of bindings, each of which will send q to a symbolic state in the symbolic control-flow graph. Using the definition in Figure 4.4, we have $\mathsf{support}_\forall(\mathsf{changes}(y)) = \{\sigma_3\}$. Hence, any concrete state in a dynamic run over this symbolic control-flow graph containing the symbolic state σ_3 will be identified by the quantifier. The final set $\mathcal{B}_\varphi$ is then $\{[q \mapsto \sigma_3]\}$.

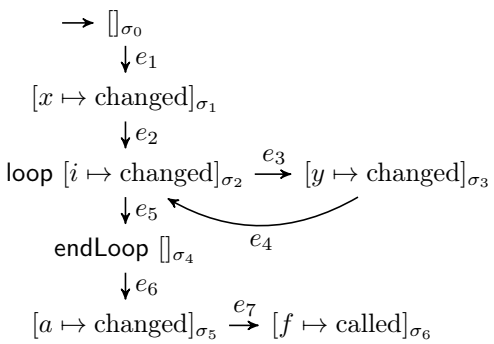


Figure 4.5: An example symbolic control-flow graph for demonstration of static binding computation.

It remains to determine which symbolic states reachable from σ_3 will be associated with the concrete states at both ends of the transition required to decide whether $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$ holds. Our next step is to introduce a general approach.

4.7.3 Inspecting Atoms

The concrete states and transitions needed to decide whether the quantifier-free part of a specification is true given a specific binding are identified using the eval function. Hence, our goal is now to inspect the atoms of a specification in order to determine the symbolic states that could be associated with any concrete states/transitions that eval could identify.

Symbolic Support for Atoms

The definition, given in Figure 4.6, of the symbolic support needed for the quantifier-free part of a specification is more involved than for quantifier predicates because there are more cases to cover, and with more complex structure. For example, the next temporal operator can be arbitrarily nested, so computing the symbolic support for this requires recursion. Further, mixed atoms require computation of the symbolic support for each expression, which themselves can use nested temporal operators.

We now describe the intuition involved in this extended definition of symbolic support. The rule for $\mathsf{support}(\theta, s_i)$ is straightforward, and requires that a symbolic state is in the symbolic support of a variable if the same symbolic state is in the symbolic support of the predicate that generates values of that variable. This criterion is expressed by the condition $\theta(s_i) = \sigma$ for σ the symbolic state whose containment in the symbolic support we are checking. $\mathsf{support}(\theta, \mathsf{dest}(T))$ and $\mathsf{support}(\theta, \mathsf{source}(T))$ involve recursion on potentially complex transition terms T. For example, in CFTL one can write $\mathsf{dest}(t)$, but one can also write $\mathsf{dest}(\mathsf{next}(\mathsf{next}(q, \mathsf{changes}(x)), \mathsf{calls}(f)))$ to refer to "the resulting concrete state of the next call of f, after the next change to x, after the state stored in q".

The symbolic support rules for the next operator are more involved in order to deal with arbitrary nesting of operators. We first have the cases that deal with the next operator being applied to a variable s_i, in which case the symbolic support we have to compute initially is just $\mathsf{support}(\Gamma_i)$, for which we have already given a construction. If the next operator is applied to a chain of temporal operators, then we use X to represent such a chain. The recursive base case is reached once X is just a variable, rather than a temporal operator applied to a variable.

If we again take the CFTL specification $\forall q \in \mathsf{changes}(y) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$ with the set of static bindings that we computed earlier, $\mathcal{B}_\varphi = \{[q \mapsto \sigma_3]\}$, then we can apply the extension of symbolic support to determine the symbolic states in the reachable neighbourhood of σ_3 in its symbolic control-flow graph. We apply support to all parts of the quantifier-free part of the specification generated by S and T, which is in this case just one part, $\mathsf{next}(q, \mathsf{calls}(f))$, and the result is $\mathsf{support}(\theta, \mathsf{next}(q, \mathsf{calls}(f))) = \{\langle \sigma_5, \sigma_6 \rangle\}$ using $\theta(q) = \sigma_3$.

We now define the notion of *lifting* an atom in A_φ into the *static context*, by which we mean the process of computing the symbolic support of an atom with respect to the symbolic control-flow graph, a statically-computable structure. In Figure 4.7, we define the function $\mathsf{lift}(\alpha)$ for $\alpha \in A_\varphi$ which takes α and gives the set of symbolic states and edges that may correspond

$\sigma \in \mathsf{support}(\theta, s_i)$ if $\theta(s_i) = \sigma$

$\sigma \in \mathsf{support}(\theta, \mathsf{next}(s_i, \mathsf{changes}(x))$ if
$\sigma(x) =$ changed and
$\exists \sigma_1 \in \mathsf{support}(\theta, s_i)$ such that there is a
path π from σ_1 to σ and there is no $\sigma_2 \in \pi$ with $\sigma_2 \neq \sigma$ and $\sigma_2(x) =$ changed

$\langle \sigma_1, \sigma_2 \rangle \in \mathsf{support}(\theta, \mathsf{next}(s_i, \mathsf{calls}(f))$ if
$\sigma_2(f) =$ called and $\exists \sigma' \in \mathsf{support}(\theta, s_i)$ such that there
is a path π from σ' to σ_1 and $\langle \sigma_1, \sigma_2 \rangle \in E$
and there is no $\sigma'' \in \pi$ with $\sigma''(f) =$ called

$\sigma' \in \mathsf{support}(\theta, \mathsf{dest}(T))$ if there is $\langle \sigma, \sigma' \rangle \in \mathsf{support}(\theta, T)$

$\sigma \in \mathsf{support}(\theta, \mathsf{source}(T))$ if there is $\langle \sigma, \sigma' \rangle \in \mathsf{support}(\theta, T)$

$\sigma \in \mathsf{support}(\theta, \mathsf{next}(X, \mathsf{changes}(x))$ if
$\sigma(x) =$ changed and
$\exists \sigma_1 \in \mathsf{support}(\theta, X)$ such that there is a
path π from σ_1 to σ and there is no $\sigma_2 \in \pi$ with $\sigma_2 \neq \sigma$ and $\sigma_2(x) =$ changed

$\langle \sigma_1, \sigma_2 \rangle \in \mathsf{support}(\theta, \mathsf{next}(X, \mathsf{calls}(f))$ if
$\sigma_2(f) =$ called and $\exists \sigma' \in \mathsf{support}(\theta, X)$ such that there
is a path π from σ' to σ_1 and $\langle \sigma_1, \sigma_2 \rangle \in E$
and there is no $\sigma'' \in \pi$ with $\sigma''(f) =$ called

Figure 4.6: The definition of symbolic support for the quantifier-free part of CFTL specifications.

$$
\begin{aligned}
\mathsf{lift}(\theta, expr_1 \ cmp \ expr_2) &= \{[expr_1 \mapsto \mathsf{support}(\theta, \mathsf{temp}(expr_1)), \\
&\qquad expr_2 \mapsto \mathsf{support}(\theta, \mathsf{temp}(expr_2))]\} \\
\mathsf{lift}(\theta, \mathsf{timeBetween}(S_1, S_2) \ cmp \ n) &= \{[S_1 \mapsto \mathsf{support}(\theta, S_1), S_2 \mapsto \mathsf{support}(\theta, S_2)]\} \\
\mathsf{lift}(\theta, \phi_S) &= \mathsf{support}(\theta, \mathsf{temp}(\phi_S)) \\
\mathsf{lift}(\theta, \phi_T) &= \mathsf{support}(\theta, \mathsf{temp}(\phi_T))
\end{aligned}
$$

Figure 4.7: The definition of the lift function for an atom $\alpha \in A_\varphi$.

Algorithm 5 Construct the function $\mathcal{H}_\varphi$ with a symbolic control-flow graph implicit and a set of static bindings $\mathcal{B}_\varphi$.

```
1: H_φ ← []
2: %We assign an arbitrary index to each B_φ and A_φ.
3: for θ ∈ B_φ with index i_θ do
4:    for α ∈ A_φ with index i_α do
5:       H_φ⟨i_θ, i_α⟩ ← lift(θ, α)
```

to concrete states and transitions in a dynamic run. The lift function makes use of the temp function, defined in Section 3.4.2 on page 66.

For mixed atoms, lifting results in a set containing a single map sending each of the two parts of the atom to their respective symbolic supports. For normal atoms, lifting results in a set containing symbolic states or edges from the symbolic control-flow graph.

4.7.4 Dividing up Inst

The theory developed so far for instrumentation needs to be brought together into a single instrumentation process. Our goal is to compute the set Inst of *instrumentation points* before monitoring (ie, before the program under scrutiny runs) and then modify our existing algorithm to use Inst.

The first step in doing this involves looking more closely at the definitions of the lift and support functions. We have defined lift and support in a way that requires a static binding. This is to allow the set Inst to be computed in subsets with respect to 1) static bindings and then 2) atoms. By doing this we get the situation where, if a concrete state is processed by the monitoring algorithm, a lookup can be performed to determine the bindings from $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ to which the concrete state contributes. Further, we can group by atom to allow the part of the formula tree that must be updated for each binding to be found quickly. Algorithm 5 details the construction of a structure $\mathcal{H}_\varphi$ that forms this division into subsets of Inst. The structure $\mathcal{H}_\varphi$ is a map from pairs $\langle i_\theta, i_\alpha \rangle$ of indices to the results of lift. i_θ and i_α are indices of a binding and an atom in $\mathcal{B}_\varphi$ and A_φ respectively. We can choose arbitrary indices, as long as we are consistent; in practice, indices of bindings are derived from the order in which the bindings are computed, and indices of atoms can be derived from an inorder traversal of the formula tree.

Before we decide how to use $\mathcal{H}_\varphi$ to improve our monitoring algorithm, we observe that the images of $\langle i_\theta, i_\alpha \rangle$ under $\mathcal{H}_\varphi$ are not necessarily disjoint. One can see this by considering the simple program

```
1  a = 10
2  a = 20
3  f(a)
```

monitoring with respect to the CFTL specification

$$\forall q \in \mathsf{changes}(a) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 1).$$

In this case, the set $\mathcal{B}_\varphi$ of static bindings would contain the symbolic states generated by lines 1 and 2. For each of these static bindings, Algorithm 5 would identify the call to f on line 3 as being relevant for the atom $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 1)$. Letting the static bindings have indices i_θ and i'_θ in $\mathcal{B}_\varphi$ respectively, and letting $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) \in (0, 1)$ have index 0 in A_φ, the edge in the symbolic control-flow graph that would correspond to the call to f on line 3 would necessarily appear in both $\mathcal{H}_\varphi\langle i_\theta, 0\rangle$ and $\mathcal{H}_\varphi\langle i'_\theta, 0\rangle$. This lack of disjointness arises because the traversal of the symbolic control-flow graph from each of the static bindings results in the same edge being found.

4.7.5 Using $\mathcal{H}_\varphi$ for Lookup

For each concrete state that is processed, the way in which we use $\mathcal{H}_\varphi$ to perform lookup in the monitoring algorithm will involve determining first the static bindings, and then the atoms to which the relevant symbolic state corresponds. Taking $\mathcal{H}_\varphi$ without any modifications would require a search through this structure. If there are many static bindings or atoms, this search could become inefficient and so we may as well not attempt to optimise the algorithm. Therefore, it is clear that a step is needed to transform $\mathcal{H}_\varphi$ into a structure over which search will be efficient. Before introducing this step, we highlight that we would increase granularity of $\mathcal{H}_\varphi$ by adding an element to the $\langle i_\theta, i_\alpha \rangle$ pair to distinguish between instrumentation points for the left and right-hand sides of mixed atoms. However, from our experience in the practical setting this is rarely needed.

Our additional step is to construct an *inverse* of the map $\mathcal{H}_\varphi\langle i_\theta, i_\alpha \rangle$, so we instead have a map from elements given by the lift function (so either symbolic states in the case of normal atoms, or maps in the case of mixed atoms) to pairs containing the static binding index and the atom index. Such an inversion would allow us determine the static binding/atom index pairs associated with a given symbolic state/edge from the symbolic control-flow graph.

Now, to invert $\mathcal{H}_\varphi$, we must address the fact that it is not necessarily a bijection, so its inverse may not exist. To see that it is not a bijection, one needs only to consider the simple program from earlier in which $\mathcal{H}_\varphi\langle i_\theta, 0\rangle = \mathcal{H}_\varphi\langle i'_\theta, 0\rangle$, so $\mathcal{H}_\varphi$ may not be a bijection because it may not be injective. With this in mind, we define the inverse by

$$\mathcal{H}_\varphi^{-1}(v) = \{\langle i_\theta, i_\alpha \rangle : v \in \mathcal{H}_\varphi\langle i_\theta, i_\alpha \rangle\}$$

where v is either a symbolic state/edge from the symbolic control-flow graph in the case of

a normal atom, or a map given by lift in the case of a mixed atom. Therefore, $\mathcal{H}_\varphi^{-1}$ allows us to refer to the set of pairs of binding and atom indices that $\mathcal{H}_\varphi$ would send to sets of 1) symbolic states or edges (in the case of a normal atom); or 2) maps from expressions to symbolic states/edges (in the case of a mixed atom). We denote this set of pairs, for a given symbolic state, edge or map v, by $\mathcal{H}_\varphi^{-1}(v)$.

One can also compute a map from symbolic states/edges to the set of static bindings in whose image they are contained. We will denote this map by $\mathcal{R}_\varphi$, and define it by

$$\mathcal{R}_\varphi(v) = \{\theta : \theta(q) = v \text{ for some } q \in \mathsf{vars}(\varphi) \text{ and } \theta \in \mathcal{B}_\varphi\}.$$

With these two structures, we would be able to decide:

- Which bindings from $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ are relevant (via static bindings) given a symbolic state whose corresponding concrete state could contribute to the truth value reached for a binding. For this, we would use $\mathcal{H}_\varphi^{-1}$.
- Whether a binding should be added to $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ based on a specific symbolic state. For example, if a CFTL specification has the quantifier $\forall q \in \mathsf{changes}(a)$, then a symbolic state σ with $\sigma(a) =$ changed must result in the generation of a new binding in $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$. For this, we would use $\mathcal{R}_\varphi$.

With these structures developed, we are in a position to optimise Algorithm 2.

4.8 Efficient Monitoring

We consider Algorithm 2 again. Our goal is to optimise the process we go through for each concrete state that we process from a dynamic run. With the current version of the algorithm, we go through two main stages:

- Construction of $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$.
- Update of the formula tree associated with each $\beta \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$.

The inefficiency in our current construction of $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ lies in the fact that every concrete state observed must be evaluated for agreement with one of the quantifier predicates. This, as we have already seen, consists only of checking the symbolic state contained in the concrete state.

4.8.1 Generating Less Verbose Dynamic Runs

Our first step in optimisation is to safely remove concrete states from the dynamic run which are needed by neither the quantifiers of a CFTL specification, nor the atoms. Firstly, $\mathcal{R}_\varphi$ gives us a way to decide which symbolic states will be associated with concrete states/transitions that are identified by quantifiers. Hence, since $\mathcal{R}_\varphi$ can be computed with only a symbolic control-flow graph and CFTL specification, instrumentation can lead to the generation at runtime of

a dynamic run that does not include concrete states/transitions that are not relevant to any quantifiers. We call this policy *quantifier-driven filtering*. However, by throwing away all of the concrete states that are not relevant with respect to $\mathcal{R}_\varphi$ we could lose the concrete states that are needed to evaluate the quantifier-free part of the CFTL specification at each binding in $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$. To avoid throwing these away, we apply the following policy, which we call *atom-driven filtering*:

- Keep concrete states $\langle t, \sigma, \tau \rangle$ such that $\sigma \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$. This is the case if the symbolic state σ was identified based on a normal atom.
- Keep transitions $\langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle$ such that $\langle \sigma, \sigma' \rangle \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$. Similarly to the point above, this is the case if the edge $\langle \sigma, \sigma' \rangle$ was identified based on a normal atom.
- Keep concrete states/transitions associated with symbolic states/edges which are found in the image of some expression from a mixed atom under a map $m \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$. While, for normal atoms, single symbolic states or edges are identified (and are therefore found in $\mathsf{dom}(\mathcal{H}_\varphi^{-1})$), for mixed atoms, maps are constructed that send each expression in the atom to a symbolic state or edge. Hence, such maps are found in $\mathsf{dom}(\mathcal{H}_\varphi^{-1})$ (see Section 4.7.3).

To safely remove from a dynamic run the concrete states that will not be needed to check a CFTL specification φ, we therefore apply the two sets of conditions. We apply those above to be sure not to throw away a concrete state that is needed to evaluate the quantifier-free part of the CFTL specification, and we do not throw away any concrete state whose symbolic state is found in $\mathsf{dom}(\mathcal{R}_\varphi)$. Ultimately, we call elements of the set

$$\mathsf{Inst} = \mathsf{dom}(\mathcal{R}_\varphi) \cup \{v \in \mathsf{dom}(\mathcal{H}_\varphi^{-1}) : v \text{ is a symbolic state or edge}\} \cup$$

$$\left(\bigcup_{m \in \mathsf{dom}(\mathcal{H}_\varphi^{-1}) \text{ and } m \text{ is a map}} \mathsf{range}(m)\right)$$

the *instrumentation points* derived from a symbolic control-flow graph with respect to the CFTL specification φ. Here, we say again that m is a map, constructed with respect to some mixed atom, sending each expression in that mixed atom to a set of symbolic states/edges. Given the set Inst, we say that any symbolic state or edge that is not in Inst is *redundant with respect to* φ.

We now prove that application of these rules to filter a dynamic run does not compromise the monitoring process. In particular, we prove this in the context of Algorithm 2 because no optimisations have yet been introduced to lookup.

Theorem 4. *For dynamic run $\mathcal{D}$ and another dynamic run $\mathcal{D}'$, which is $\mathcal{D}$ modified with respect to* quantifier *and* atom-driven filtering*, $\mathcal{D} \models \varphi$ if and only if $\mathcal{D}' \models \varphi$.*

Proof. We prove the *if* direction. Without loss of generality, we fix a concrete state $s = \langle t, \sigma, \tau \rangle$ (with σ not an instrumentation point) in $\mathcal{D}$ but not in $\mathcal{D}'$, hence has been removed by modification with respect to either quantifier or atom-driven filtering. We then suppose that $\mathcal{D}' \not\models \varphi$ with the intention of deriving a contradiction.

Since σ is not an instrumentation point, we have that $\sigma \notin \mathsf{dom}(\mathcal{R}_\varphi)$ (σ does not contribute to a binding) and $\sigma \notin \mathsf{dom}(\mathcal{H}_\varphi^{-1})$ (σ is not held by any concrete state that contributes to evaluation of a formula tree).

If the verdict changed from satisfaction in $\mathcal{D}$ to violation in $\mathcal{D}'$, then $\sigma \notin \mathsf{dom}(\mathcal{R}_\varphi)$ can hold since removal of bindings cannot lead to a change in verdict. This can be seen by supposing that removal of an element from $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ *can* change a verdict, which immediately leads us to assert that there must be some other binding in $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, remaining after the removal of σ, that causes the violation in $\mathcal{D}'$. If this were to be the case, then that binding would also cause the violation in $\mathcal{D}$, but it does not so cannot exist.

The verdict changing from satisfaction to violation means that a concrete state that was needed was removed, resulting in a temporal operator defaulting to *false*. However, this is a contradiction since we assumed that $\sigma \notin \mathsf{dom}(\mathcal{H}_\varphi^{-1})$. Therefore, it must be the case that $\mathcal{D}' \models \varphi$, so the result follows.

Now we prove the *only if* direction. Consider a dynamic run $\mathcal{D}'$ that has been filtered with respect to quantifier and atom-driven filtering, starting from an original dynamic run $\mathcal{D}$. Suppose that $\mathcal{D}' \models \varphi$ and that $\mathcal{D} \not\models \varphi$. We show that this cannot be the case under the assumption that $\mathcal{D}'$ was obtained from $\mathcal{D}$ via quantifier and atom-driven filtering.

If $\mathcal{D} \not\models \varphi$ and $\mathcal{D}' \models \varphi$, then the only way in which this can happen is if a concrete state s found in $\mathcal{D}$ but not in $\mathcal{D}'$ was relevant to some part of the CFTL specification φ. In this case, the absence of s in $\mathcal{D}'$ must have led a weak temporal operator (like $\mathsf{next}_W(...)$) to say that the constraint expressed by the atom α, containing the weak temporal operator, was satisfied (this is the only way that absence of information can lead to satisfaction). In the pre-filtering dynamic run $\mathcal{D}$, this absent concrete state s must then be present and, further, must cause the constraint expressed by the atom α to be violated, leading to $\mathcal{D} \not\models \varphi$.

However, if s does indeed contribute to the truth value of some atom in φ, then the instrumentation process would ensure that it stayed by including its symbolic state in the set of instrumentation points. Hence, s would also be in $\mathcal{D}'$. We therefore have a contradiction and the result follows. □

This process of removing concrete states from a dynamic run that we are sure will not be needed is ultimately known as *instrumentation* and will be shown later to take a significant engineering effort in the implementation developed at CERN.

A Remark on Instrumentation Optimality

Despite the efforts presented here, the dynamic run resulting from our filtering strategy is not guaranteed to be optimal. We do not define *optimality* of instrumentation here, but we informally say that an optimal instrumentation strategy would require no decisions from the monitoring algorithm about whether a given concrete state was actually needed. To see when the instrumentation strategy that we have presented may not be optimal, consider the program

```
a = 10
for i in range(10):
  f(a)
```

and the CFTL specification $\forall q \in \mathsf{changes}(a) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$. Even by applying our filtering strategy here, the resulting dynamic run would contain 9 transitions representing calls to f that would not be needed. One solution here is to unroll the loop. However, loop unrolling would require that we are sure that the section of control-flow to be factored out of the loop will be executed on the first iteration. This is a constraint satisfaction problem.

Ultimately, in the cases considered in this thesis, loop unrolling would not be beneficial enough to justify a complete treatment. In particular, loop unrolling would certainly result in a large optimisation if we had a loop that completed thousands of iterations and only the first was needed (if, for example, our CFTL specification used next rather than future), but in our use cases (see Chapter 7), specifications were not written over loops in a way that would benefit from such an optimisation.

4.8.2 Lookup

With our filtering mechanism, we can be sure that a filtered dynamic run will contain only concrete states that either contribute to bindings or evaluation of formula trees. This means that the dynamic run that must be processed by the monitoring algorithm contains fewer concrete states, meaning the monitoring algorithm does less work. The remaining problems to solve are those surrounding the problem of how to decide whether constraints defined by a CFTL specification are satisfied by the concrete states given in a dynamic run. More precisely, the problems are as follows:

- Given a concrete state that may contribute to a binding, how do we determine which existing bindings should be extended? For example, if we are looking at a concrete state that satisfies Γ_i and we already have a set of existing bindings, which ones do we modify to include a new entry?
- Given a concrete state that contains information that an atom in some formula tree needs, how do we determine which formula tree to update?

We now describe how $\mathcal{R}$ and $\mathcal{H}_\varphi^{-1}$ can be used to optimise the lookup required in both of the cases listed above.

Faster Lookup of Bindings

Currently, during the construction of $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, we have to iterate through all existing bindings to determine which can be extended. However, to take an example, it may be the case that we are processing a concrete state $s = \langle t, \sigma, \tau \rangle$ that satisfies the predicate for the variable q_i. If there are already bindings such that either 1) their domains include q_j for $j \geq i$; or 2) they correspond to static bindings whose range does not include σ, then we must still process these but will never extend them. We can make this process more efficient by combining the information from the symbolic control-flow graph contained by concrete states with the map $\mathcal{R}_\varphi$. The result is that we will be able to organise bindings by the static bindings to which they correspond.

To see how $\mathcal{R}_\varphi$ can be used to improve efficiency when constructing $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, we take a concrete state $s = \langle t, \sigma, \tau \rangle$ and first ask if $s \in \mathsf{dom}(\mathcal{R}_\varphi)$. If this is the case, then this means that the concrete state s must contribute to one or more bindings. We then determine $\mathcal{R}_\varphi(\sigma)$, the set containing each static binding θ for which there is some variable q such that $\theta(q) = \sigma$. The reasoning would be similar if we were to consider a transition rather than a concrete state. Given this use of $\mathcal{R}_\varphi$, we make the following modifications to Algorithm 1, resulting in Algorithm 6:

- Replace the set M by a map from static binding indices to sets of bindings derived from the dynamic run.
- Replace the final assignment of C with a loop which removes from M any partial bindings. Hence M is ultimately a map from static binding indices to the bindings derived from the dynamic run that correspond to those static bindings.
- Add a check for whether the symbolic state of curr is in $\mathsf{dom}(\mathcal{R}_\varphi)$, to decide whether it contributes to the set of bindings.

Algorithm 6 An optimised algorithm for the derivation of the set of complete bindings from the dynamic run $\mathcal{D}$ based on the quantifiers $\forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n$.

```
1:  M ← []                                   ▷ Empty map from static binding indices to sets of bindings
2:  for concrete state curr = ⟨t, σ, τ⟩ ∈ D do      ▷ Initialise M with empty set for each θ
3:    if σ ∈ dom(R_φ) then
4:      for θ ∈ R_φ(σ) do
5:        i_θ ← index of θ ∈ B_φ
6:        M(i_θ) ← ∅
7:  prev ← ⟨0, [], []⟩                                  ▷ To store the previous state
8:  for concrete state curr = ⟨t, σ, τ⟩ ∈ D do
9:    if σ ∈ dom(R_φ) then
10:     for θ ∈ R_φ(σ) do
11:       i_θ ← index of θ ∈ B_φ
12:       I_∀ ← {i : θ(q_i) = σ for q_i ∈ vars(φ)}        ▷ Loops can mean repeated entries
13:       for i_∀ ∈ I_∀ do      ▷ Add new or extended binding depending on the conditions
14:         if i_∀ = 1 then
15:           M(i_θ) ← M(i_θ) ∪ {[s_1 ↦ curr]}
16:         else if i_∀ > 1 then
17:           for β ∈ M(i_θ) do
18:             if |dom(β)| = i_∀ − 1 then
19:               M(i_θ) ← M(i_θ) ∪ {β † [s_{i_∀} ↦ curr]}
20:   %Similar in the case of a transition/edge...
21:   prev ← curr
22: %Check for complete bindings - we just need those
23: for i_θ ∈ dom(M) do
24:   for β ∈ M(i_θ) do
25:     if β is not complete then
26:       M(i_θ) ← M(i_θ)\{β}
27: return M
```

Faster Lookup of Formula Trees

With the algorithm optimised for construction of the set $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, we now make the necessary modifications to Algorithm 2. The result is Algorithm 7. We first modify the map C to include the grouping by indices of static bindings in $\mathcal{B}_\varphi$. After that, the modifications are to the main loop over the dynamic run. In particular, we check whether σ or some map m whose range contains σ is contained by $\mathsf{dom}(\mathcal{H}_\varphi^{-1})$. This distinction, from the construction of $\mathcal{H}_\varphi$, comes from the fact that mixed atoms result in maps, while normal atoms simply result in symbolic states or edges. Ultimately, this check is a direct way to decide whether the concrete state currently being processed is required by some binding in $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$; by construction of $\mathcal{H}_\varphi$, all concrete states that will be used by a formula tree correspond to some symbolic state contained there.

We consider the case in which we just find a $\sigma \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$, rather than a map m. Here, if σ is contained in the domain, then we necessarily have a set of pairs whose left element is the index of a static binding in $\mathcal{B}_\varphi$, and whose right element is the index of an atom in A_φ. Iterating through this set of pairs, we use i_θ to access only the complete bindings in C that correspond to the static binding at index i_θ. We then use i_α to make the update procedure more efficient; the original version requires search through all of the atoms in the formula tree to determine which can be evaluated with respect to the concrete state/transition observed. We can improve this by going immediately to the atom in the formula tree with index i_α.

4.8.3 Complexity of the Optimised Monitoring Algorithm

We assume that we apply the optimised Algorithm 7 to a dynamic run that has been filtered with respect to quantifiers and atoms. Deriving an exact complexity in this case is not feasible because there can be so much variation in how much of a dynamic run can be discarded by filtering. If a lot of concrete states found in a dynamic run are thrown away, and we use the new lookup approach that we now have, efficiency must improve because the input is smaller and the statically computable information from instrumentation ($\mathcal{H}_\varphi$ and $\mathcal{R}_\varphi$) allows lookup of bindings to be done much more quickly. If not much of a dynamic run is thrown away, then we get improvement solely from the faster lookup machinery.

As a final remark, we highlight that, since instrumentation information is available in the monitoring algorithm, one could use the atom index i_α to improve lookup time when updating formula trees. We do not define it here because the benefit in our use cases would not be felt, but it would be a matter of maintaining a map from atom indices to parts of the formula tree and using this to perform the update. This would remove the need for traversal and result in a large optimisation if the CFTL specifications being considered contained many atoms and propositional connectives (since this is the situation in which traversal of the formula tree is most expensive).

Algorithm 7 Optimised checking of the dynamic run $\mathcal{D}$ for satisfaction of the CFTL specification $\forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n : \phi$.

1: *%Use the optimised algorithm to construct a set of complete bindings*
2: $B \leftarrow \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 s_1 \in \Gamma_1 : \ldots : \forall_n s_n \in \Gamma_n)$
3: *%Initialise an empty map from static binding indices to bindings to formula trees*
4: $C \leftarrow []$
5: *%Initialise the map C with formula trees.*
6: **for** $i_\theta \in B$ **do**
7: **for** $\beta \in B(i_\theta)$ **do**
8: $C(i_\theta) \leftarrow C(i_\theta) \dagger [\beta \mapsto \mathsf{tree}(\varphi)]$
9: *%Iterate through $\mathcal{D}$, updating the formula trees to obtain verdicts*
10: $\mathsf{prev} \leftarrow \langle 0, [], [] \rangle$ ▷ Assign an empty concrete state
11: **for** $\mathsf{curr} = \langle t, \sigma, \tau \rangle \in \mathcal{D}$ **do** ▷ Iterate through the dynamic run
12: **if** $\sigma \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$ **then**
13: **for** $p \in \mathcal{H}_\varphi^{-1}(\sigma)$ **do** ▷ Iterate through the relevant pairs of static bindings and atoms
14: $i_\theta, i_\alpha \leftarrow p$
15: **for** (β, T) in $C(i_\theta)$ **do** ▷ Process formula trees associated with relevant bindings
16: $T' \leftarrow \mathsf{update}_\beta(T, \beta, \langle \mathsf{prev}, \mathsf{curr} \rangle)$
17: **if** $T' = \mathit{false}$ **then return** *false*
18: $C(i_\theta) \leftarrow C(i_\theta) \dagger (\beta \mapsto T')$
19: $T' \leftarrow \mathsf{update}_\beta(T, \beta, \mathsf{curr})$
20: **if** $T' = \mathit{false}$ **then return** *false*
21: $C(i_\theta) \leftarrow C(i_\theta) \dagger (\beta \mapsto T')$
22: **else if** there is a map $m \in \mathsf{dom}(\mathcal{H}_\varphi^{-1})$ with $\sigma \in \mathsf{range}(m)$ **then**
23: **for** $p \in \mathcal{H}_\varphi^{-1}(m)$ **do** ▷ The same approach is taken here as above
24: $i_\theta, i_\alpha \leftarrow p$
25: **for** (β, T) in $C(i_\theta)$ **do**
26: $T' \leftarrow \mathsf{update}_\beta(T, \beta, \langle \mathsf{prev}, \mathsf{curr} \rangle)$
27: **if** $T' = \mathit{false}$ **then return** *false*
28: $C(i_\theta) \leftarrow C(i_\theta) \dagger (\beta \mapsto T')$
29: $T' \leftarrow \mathsf{update}_\beta(T, \beta, \mathsf{curr})$
30: **if** $T' = \mathit{false}$ **then return** *false*
31: $C(i_\theta) \leftarrow C(i_\theta) \dagger (\beta \mapsto T')$
32: $\mathsf{prev} \leftarrow \mathsf{curr}$
33: *%Find the complete bindings whose formula trees have not reached a verdict*
34: *%For each such binding, we find the atoms for which a truth value has not been determined*
35: *%and decide what the truth value must be based on whether the temporal operator involved*
36: *%is normal or weak.*
37: **for** $i_\theta \in B$ **do**
38: **for** (β, T) in $C(i_\theta)$ **do**
39: **if** $T \neq \mathit{true}$ and $\mathsf{resolve}(T) = \mathit{false}$ **then**
40: **return** *false*
41: **return** *true*

4.9 Concluding CFTL

We have now introduced all of the major machinery for the construction of CFTL specifications and the subsequent instrumentation and monitoring of a program. We began by introducing the syntax and semantics of CFTL, a new temporal logic. We then introduced a first attempt at monitoring CFTL specifications given *complete* dynamic runs, and realised that by performing some static analysis we can remove a lot of work done by the monitoring algorithm. We will now move on and discuss the problem of *explaining why* a program did or did not satisfy a specification expressed in CFTL.

Chapter 5

Explaining Violations

This thesis has so far addressed the problem of deciding whether a program satisfies a property at runtime. In this chapter, based largely on the material presented at the Runtime Verification 2019 and 2020 conferences [DR19a, DHJ+20], we address the more challenging problem of finding some indication, which we will call an *explanation*, of *why* a program run did not satisfy a property.

5.1 What Constitutes an Explanation

Given the low level of abstraction of Control-Flow Temporal Logic, we have the opportunity to base our explanation approach on the structures closely associated with the source code, such as paths taken through code, values held by variables and call trees. The case that was most useful in our case study in Chapter 7 was the analysis of program paths, since there are multiple cases in that system where the behaviour can be characterised by the path taken through the code at runtime. Further, in Chapter 7 we also partially reconstruct call trees in order to connect information from dynamic runs generated by multiple functions.

In light of our case study, this chapter focuses on the analysis of paths taken through source code by dynamic runs. The notion of an *explanation* will take the form of the difference that can be found between paths taken by multiple dynamic runs, and how this difference relates to verdicts generated by monitoring for CFTL specifications.

The complexity of the problem of determining such explanations lies not only in the task of developing theory that allows the generation of explanations, but also in generating explanations that are useful to software engineers. In general, given a low-level specification in CFTL written about a program, the ideal scenario for a software engineer is to observe a violation of their specification and, hopefully quickly, be given some possible indication of the cause. Such a cause could include problematic variable values, regions of code (via control-flow), chains of function calls, and more. Hence, the problem of determining an explanation depends also on the domain in which the software being monitored operates. The explanation approach that we present, that of looking at program paths, will focus on information that is easy to obtain at the source code level of Python programs. We will also discuss analysis that is possible in

the interprocedural and inter-machine settings.

5.2 Developing our Explanation Approach

We give a brief overview of the components of our explanation approach. In this chapter, we give the theoretical foundations and leave practical considerations to Chapter 6.

5.2.1 Program Paths and their Relevance to CFTL

One of the fundamental notions of the explanation approach that we present is that of a *program path*. We will present paths through programs as paths through symbolic control-flow graphs; this will allow us to develop a powerful comparison method.

Further, since each concrete state in a dynamic run corresponds to a symbolic state in a symbolic control-flow graph, and symbolic states contain program points, one can easily imagine that a program path leading to a specific concrete state is simply a path through the symbolic control-flow graph up to the relevant symbolic state. However, determining this path requires information from a dynamic run because program paths traverse constructs such as loops whose number of iterations often cannot be determined in the static context.

Part of our explanation approach is additional instrumentation to allow the exact program path to be determined. This process involves 1) a straightforward determination of the symbolic states to include in the set of instrumentation points; and 2) *reconstruction* of the exact program path taken given a dynamic run with additional concrete states included based on this instrumentation.

We then construct explanations by introducing a new approach to comparing program paths. This approach allows us to determine the regions of source code that were executed in the cases in which a CFTL specification was violated, and not when it was satisfied. Further, our combination of program paths with the concrete states to which they lead allows us to use the results of path comparison to explain verdicts obtained by monitoring CFTL specifications.

Attaching Timing Information to Paths

Our initial work on program path comparison for explanation can be extended by attaching timing information to program paths. This modification helps us in the case that paths are shown be the same in certain regions. For example, we may have paths between two concrete states which are the same except from the number of iterations of two loops that they each perform. In this case, we might be interested in determining the extra time taken by one of the paths to complete additional iterations of the loops. Such information may help to determine the problematic code if, for example, we are placing a constraint over the time taken to get from one concrete state to another. This extension is not considered in this thesis because there was never a practical need for it.

5.2.2 Secondary Explanation Approaches

Secondary to our program path-based approach, we present two additional approaches for determining causes of problematic program behaviour in the case that analysis of program paths does not yield anything useful.

First, we describe an approach involving partially reconstructing the call tree based on which functions across a system were being monitored at runtime. By doing this, we can traverse the call tree to determine which CFTL specifications may have been affected by others lower down in the call tree.

Second, we highlight that the concrete states in a dynamic run store information about the values held by variables. Our final approach is to do straightforward analyses on these quantities to determine whether certain values are more likely to cause verdicts. However, though this approach is a straightforward extension of the existing machinery, we do not use it in the case study presented in Chapter 7.

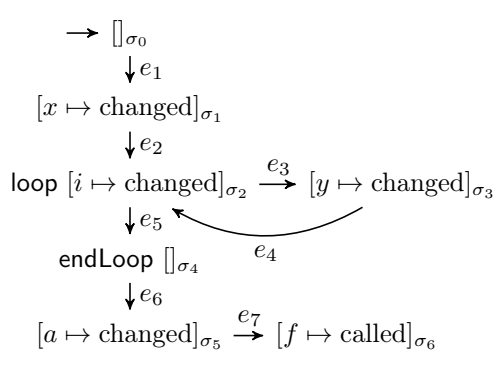
5.3 Program Paths

We begin developing our explanation approach by considering program paths as paths through symbolic control-flow graphs. Recall that such a path is a sequence of edges $\langle \sigma, \sigma' \rangle$ through the symbolic control-flow graph and, in addition, these graphs capture the control-flow present in a program. For example, if a conditional is present, the symbolic control-flow graph will capture the divergence and subsequence convergence. Loops are also captured by cycles.

Since, in this thesis, the model of program runs used is the notion of a dynamic run, we consider a most-general, complete dynamic run over a symbolic control-flow graph and then the path $\pi = \langle \sigma_1, \sigma_2 \rangle, \ldots, \langle \sigma_{n-1}, \sigma_n \rangle$ with σ_n final, that is, no other symbolic states are reachable from it in the symbolic control-flow graph. The path π is what we will consider as a *program path*, that is, a sequence of edges through a symbolic control-flow graph which correspond to the exact path taken by a dynamic run.

For now we will not consider these program paths in the context of CFTL, rather just as representations of program runs that we can perform operations on. We will introduce our approach to performing comparison on program paths and then combine it with CFTL specifications to form *explanations*. These *explanations* will consist of indications of problematic regions of code by comparing paths taken across multiple dynamic runs. If we consider a system in which there are multiple functions (each of which generates a dynamic run when it is executed), an example of how problematic regions of code could be identified is either leading up to a problematic function call in a dynamic run, or through the entire dynamic run of a function that is called during a dynamic run higher in the call stack.

As a final remark, we will first consider dynamic runs that are most-general, corresponding to recording everything from a program run. In this case, the problem of reconstructing the exact path taken through the symbolic control-flow graph is straightforward. However, since a most-general dynamic run represents recording everything during a run of a program, we cannot assume that we will have most-general dynamic runs with which to form our explanations. Hence, we must address the problem of how to modify our instrumentation strategy so that



$$\begin{gathered}\langle 0.1, \sigma_0, [] \rangle, \\ \langle 0.2, \sigma_1, [x \mapsto 3] \rangle, \\ \langle 0.3, \sigma_2, [i \mapsto 0] \rangle, \\ \langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle, \\ \langle 0.9, \sigma_2, [i \mapsto 1, y \mapsto 0] \rangle, \\ \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle, \\ \langle 1.5, \sigma_2, [i \mapsto 2, y \mapsto 1] \rangle, \\ \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle, \\ \langle 1.7, \sigma_5, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle, \\ \langle 2.5, \sigma_6, [i \mapsto 2, y \mapsto 1, a \mapsto 1] \rangle\end{gathered}$$

Figure 5.1: An example, repeated from Section 3.3.3, of a most-general dynamic run over an SCFG.

$$\mathsf{branching}(\mathcal{D}_p) = \left\{ \begin{array}{c} \{\langle 0.6, \sigma_3, [i \mapsto 0, y \mapsto 0] \rangle, \\ \langle 1.4, \sigma_3, [i \mapsto 1, y \mapsto 1] \rangle, \\ \langle 1.55, \sigma_4, [i \mapsto 2, y \mapsto 1] \rangle \end{array} \right\}$$

Figure 5.2: The set $\mathsf{branching}(\mathcal{D}_p)$ derived from the dynamic run given in Figure 5.1.

the path followed by a dynamic run that is not most-general can still be reconstructed without inducing too much overhead.

5.3.1 Dynamic Runs as Program Paths

Take a complete, most-general dynamic run $\mathcal{D} = s_1 \dots s_n$ over the symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ of the program P. We would like to know the exact path taken by this dynamic run through $\mathsf{SCFG}(P)$, that is, we would like to know the sequence of edges $\langle \sigma, \sigma' \rangle \in E$ that were traversed to form $\mathcal{D}$. With a most-general dynamic run this is straightforward, as can be seen in the example in Figure 5.1. Reconstructing the path for this dynamic run is simply a matter of iterating through it to yield

$$\pi = \langle \sigma_0, \sigma_1 \rangle, \langle \sigma_1, \sigma_2 \rangle, \langle \sigma_2, \sigma_3 \rangle, \langle \sigma_3, \sigma_2 \rangle, \langle \sigma_2, \sigma_3 \rangle, \langle \sigma_3, \sigma_2 \rangle, \langle \sigma_2, \sigma_4 \rangle, \langle \sigma_4, \sigma_5 \rangle, \langle \sigma_5, \sigma_6 \rangle.$$

Further, this path will include information about loop iterations completed since $\mathcal{D}$ was generated by a real program run. However, for most-general dynamic runs, despite the lack of difficulty in solving the path reconstruction problem, generating such dynamic runs from a real system is not feasible; the set Inst constructed during instrumentation would have to be V from $\mathsf{SCFG}(P)$ and, in practice, this means taking some measurement at *every* statement in the program. This is clearly not feasible.

If one needs only to inspect the path taken by a dynamic run (and not check whether it satisfies some CFTL specification), then our approach is to use a separate instrumentation process to throw away concrete states that are not needed to determine the path. The set of such concrete states can be determined by first identifying all symbolic states in the symbolic control-flow graph whose out-degree is greater than 1, and then selecting the destination symbolic states

of the outgoing edges of each of those states. For example, if a symbolic control-flow graph were to contain a conditional, then we would keep all concrete states whose symbolic state was one of the first symbolic states in one of the clauses of the conditional. Further, if a symbolic control-flow graph were to contain a loop, then we would keep the concrete states containing the first symbolic state inside the loop body, along with the concrete states containing the loop exit symbolic state. This policy allows us to reconstruct a path by traversing a symbolic control-flow graph and using the information kept in the dynamic run to decide on which path to take whenever we encounter a symbolic state with multiple outgoing edges.

We call a dynamic run whose path through its symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ can be determined (because its concrete states have been filtered based on the instrumentation policy described above) a *branch-aware* dynamic run and define it formally as such:

Definition 5. *A dynamic run $\mathcal{D}_b$ based on a program P is a* branch-aware dynamic run *if between any two consecutive concrete states $\langle t, \sigma, \tau \rangle$ and $\langle t', \sigma', \tau' \rangle$, there is a single path π from σ to σ' in $\mathsf{SCFG}(P)$, such that in π the only appearance of σ is in the path's first edge, $\langle \sigma, \sigma_1 \rangle$ for some $\sigma_1 \neq \sigma$, and the only appearance of σ' is in the path's final edge, $\langle \sigma_2, \sigma' \rangle$ for some $\sigma_2 \neq \sigma'$.*

We denote by $\mathsf{branching}(\mathcal{D}_p)$ the set of concrete states that our instrumentation policy identifies as required for branch-awareness. Notice that, if the program whose runs we consider contains no branching, possibly $\mathsf{branching}(\mathcal{D}_p) = \emptyset$ (in the case that there are no loops or conditional blocks, ie, no opportunities for program execution to take different paths during each program run). We give an example of a set $\mathsf{branching}(\mathcal{D}_p)$ in Figure 5.2. We highlight that our instrumentation approach can determine more symbolic states than are necessary for a dynamic run to remain branch-aware, but that it admits Algorithm 8 for reconstruction of paths. This algorithm only needs to process the symbolic states that are actually in the path to be reconstructed, since the symbolic states indicated by $\mathsf{branching}(\mathcal{D}_p)$ can be used whenever there are multiple edges to traverse out of a symbolic state.

5.3.2 Reconstructing Paths after Instrumentation

We will now consider the problem of path reconstruction in the context of CFTL specifications, especially when instrumentation has been performed based on the CFTL specification. Recall that instrumentation with respect to a CFTL specification φ is a combination of inspecting the quantifier sequence and the quantifier-free part of φ. In particular, it involves removing the concrete states from a dynamic run whose symbolic states have been determined statically not to be useful. Consider the specification

$$\forall c \in \mathsf{calls}(f) : \mathsf{timeBetween}(\mathsf{source}(c), \mathsf{dest}(\mathsf{next}(c, \mathsf{calls}(g)))) < 0.5$$

that expresses the requirement that no more than 0.5 seconds should pass between the start of each call to f and the end of the subsequent call to g. Instrumentation for this specification would identify edges in the symbolic control-flow graph corresponding to calls to f and g. We

Algorithm 8 Reconstruct the path taken by a branch-aware dynamic run $\mathcal{D}_b$ through a symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$.

```
1: π ← ⟨⟩                                                    ▷ Initialise an empty path
2: branchingIndex ← 0   ▷ Initialise the index of the element of branching(D_b) to be used next
3: curr ← v_s                                  ▷ Initialise the current symbolic state to be maintained
4: while branchingIndex < |branching(D_b)| do ▷ Iterate until there are no symbolic states left
5:   support ← σ such that L(branchingIndex) = ⟨t, σ, m⟩
6:   if ⟨curr, support⟩ ∈ outgoing(curr) then                ▷ We found a branch we should take
7:     π += ⟨curr, support⟩
8:     curr ← support
9:     branchingIndex += 1
10:  else                                  ▷ No branching - keep following the path that we're on
11:    σ' ← the σ such that there is ⟨curr, σ⟩ ∈ E
12:    π += ⟨curr, σ'⟩
13:    curr ← σ'
```

would run into problems with path reconstruction for the resulting dynamic run if there were any divergence/convergence in control-flow between the two calls, where no symbolic states or edges are identified by instrumentation. In this case, we would remove concrete states from the dynamic run which were needed to identify the exact path taken through control-flow. We call this the *multiple path problem* and our solution is to apply the additional instrumentation discussed earlier at branching points, hence constructing a possibly larger (but still conservative) set of instrumentation points. With the additional concrete states in a dynamic run that would result from this instrumentation, we could perform path reconstruction.

With this in mind, we extend our definition of *branch-aware* dynamic runs to *φ-branch-aware* dynamic runs, and say that a dynamic run $\mathcal{D}$ is such if there is no concrete state that is redundant with respect to φ and whose removal would not prevent $\mathcal{D}$ from being branch-aware. The intuition here is that, if a concrete state is in a dynamic run but is not needed to check φ *and* does not contribute to branch-awareness, then it can be removed.

To conclude our discussion on making dynamic runs branch-aware, we present Algorithm 8 that reconstructs the path through a symbolic control-flow graph taken by a branch-aware dynamic run. This algorithm takes a branch-aware dynamic run with n concrete states and a symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$.

In order to ensure that this reconstruction will work, we assume that dynamic runs generated by program executions cannot have multiple concrete states with the same time (since concrete states are *induced* by statements in a program executed at runtime, meaning no statement's execution can be instantaneous). Finally, the function $\mathsf{outgoing}(\sigma)$ gives the set of edges $\{\langle \sigma, \sigma' \rangle : \sigma' \in V \text{ and } \langle \sigma, \sigma' \rangle \in E\}$.

The intuition of the algorithm is that we follow edges in the symbolic control-flow graph until we arrive at a symbolic state at which branching occurs. At this point, we use $\mathsf{branching}(\mathcal{D}_p)$ to decide on which direction to take. In order to refer to the relevant concrete state in $\mathsf{branching}(\mathcal{D}_p)$, we use an index $\mathsf{branchingIndex}$. However, in order to be able to use an index to refer to concrete states in this set, we must define an ordering with which we can construct a labelling of the set:

We introduce the ordering $<_t$ on $\mathsf{branching}(\mathcal{D}_p)$ to be such that, for $s, s' \in \mathsf{branching}(\mathcal{D}_p)$,

$s <_t s' \iff \mathsf{time}(s) < \mathsf{time}(s')$. Using this ordering, we construct the labelling L such that $L(0)$ gives the concrete state in a dynamic run that is minimal with respect to $<_t$ (ie, with respect to time) and $L(i)$ gives the i^{th} concrete state.

5.4 Determining when Violation Occurred

We now turn our attention to the idea of determining a concrete state in a dynamic run at which the verdict turned to *false*. This concrete state will give us a position in the dynamic run at which to focus our explanation approach.

In order to be able to identify such a concrete state, we need to introduce a *partial semantics*. This semantics will assume that we have monitored the dynamic run and obtained a false verdict. Further, it will enable one to *replay* the violating complete dynamic run until the prefix of it is reached that generated failure. We highlight that, while typical use of a partial semantics is to obtain a verdict as soon as possible given a large input sequence of events, the dynamic runs that we deal with do not contain many concrete states, and so there is no need to obtain a verdict before the entire dynamic run has been observed. Instead, we introduced a partial semantics as a way to define a point of failure given a dynamic run that has already been fully processed by the total semantics.

Finally, the partial semantics will hold the *verdict impartiality* [BFFR18] property, that is, *true* can never be declared for a partial dynamic run because extensions to a complete dynamic run may introduce new concrete states that lead to violations. A *false* verdict will be possible, since CFTL specifications are universally quantified so can be violated after processing a finite amount of information.

5.4.1 A Truth Domain for a Partial Semantics

Our total semantics has a truth domain $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$; due to the existence of weak and normal temporal operators and how the semantics deals with them, a value in $\mathbb{B}$ must be declared. The requirement that our semantics over partial dynamic runs is impartial means we must have another verdict to declare instead of true, since we can never safely declare this if we don't have a complete dynamic run (another concrete state can always be processed that causes the verdict to change to false). We introduce the verdict trueSoFar to be a weaker version of true that is allowed to be changed if more information is processed. Further, the behaviour of our weak and normal temporal operators must be handled differently. For example, in the total semantics we declare true if an atom uses a weak temporal operator for which no concrete state/transition is found and false in the case of a normal operator. However, in the case of a partial dynamic run, a temporal operator such as next not being able to find a relevant concrete state or transition could mean that we have to process some information from the future (ie, an extension of the partial dynamic run to a complete one). Hence, we need a verdict to handle the case where we are waiting to process some information in the future. We introduce the notSure verdict.

It then follows that, to hold the impartiality property, the partial semantics must have a 3-valued truth domain $\{\mathsf{trueSoFar}, \mathsf{notSure}, \mathsf{false}\}$, where the false here is the same as that

$$\mathsf{trueSoFar} \sqcup \mathsf{trueSoFar} = \mathsf{trueSoFar} \qquad \mathsf{trueSoFar} \sqcap \mathsf{trueSoFar} = \mathsf{trueSoFar}$$

$$\mathsf{notSure} \sqcup \mathsf{notSure} = \mathsf{notSure} \qquad \mathsf{notSure} \sqcap \mathsf{notSure} = \mathsf{notSure}$$

$$\mathsf{false} \sqcup \mathsf{false} = \mathsf{false} \qquad \mathsf{false} \sqcap \mathsf{false} = \mathsf{false}$$

$$\mathsf{trueSoFar} \sqcup \mathsf{notSure} = \mathsf{trueSoFar} \qquad \mathsf{trueSoFar} \sqcap \mathsf{notSure} = \mathsf{notSure}$$

$$\mathsf{trueSoFar} \sqcup \mathsf{false} = \mathsf{trueSoFar} \qquad \mathsf{trueSoFar} \sqcap \mathsf{false} = \mathsf{false}$$

$$\mathsf{notSure} \sqcup \mathsf{false} = \mathsf{notSure} \qquad \mathsf{notSure} \sqcap \mathsf{false} = \mathsf{false}$$

$$\neg\mathsf{trueSoFar} = \mathsf{false}$$

$$\neg\mathsf{notSure} = \mathsf{notSure}$$

$$\neg\mathsf{false} = \mathsf{trueSoFar}$$

Figure 5.3: The results of taking the least upper bound, greatest lower bound and negation of values in our 3-valued truth domain.

declared by the total semantics. Further, notSure means that, if no more information were to be obtained from the rest of the dynamic run, the final verdict yielded could still be either true or false (from the 2-valued truth domain). This is because of the presence of CFTL's weak and normal operators. Recall from Section 4.4 that, if the information required to resolve an atom is missing, its status as *weak* or *normal* determines the truth value to which it is evaluated. On the contrary, trueSoFar means that the final verdict would be true if no more information were obtained from the rest of the dynamic run.

Now, we have to address the problem of how the standard propositional connectives $(\wedge, \vee)$ will work with this truth domain. In the 2-valued truth domain case, $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ with ordering $\mathsf{false} < \mathsf{true}$, the disjunction $\vee$ is defined for two values in $\mathbb{B}$ by taking their *least upper bound*, which we denote by $\sqcup$. In this case, we have $\mathsf{true} \vee \mathsf{true} = \mathsf{true} \sqcup \mathsf{false} = \mathsf{true}$ because true is the least upper bound of true and false by the total ordering. Further, we have $\neg\mathsf{true} = \mathsf{false}$, so negation has the effect of taking the element of the truth domain that is *opposite* with respect to the total ordering.

For our 3-valued truth domain, we define the total ordering

$$\mathsf{false} < \mathsf{notSure} < \mathsf{trueSoFar}.$$

Using this, we define the least upper bound $\sqcup$, greatest lower bound $\sqcap$ and negation $\neg$ in Figure 5.3. When we give the partial semantics, this will map $\vee$ and $\wedge$ to these notions of the least upper bound and greatest lower bound on a truth domain.

5.4.2 The eval_p Function

Given our goal of determining at which point in a violating dynamic run a CFTL specification ceased to be satisfied, we must decide whether the components that we already have will work in the setting in which only prefixes of dynamic runs are available.

$$
\begin{array}{lcl}
\mathsf{eval}_p(\mathcal{D}_p, \beta, q) & = & \beta(q) \text{ if } \beta(q) \text{ is defined, } \mathsf{null} \text{ otherwise.} \\
\mathsf{eval}_p(\mathcal{D}_p, \beta, tr) & = & \beta(tr) \text{ if } \beta(tr) \text{ is defined, } \mathsf{null} \text{ otherwise.} \\
 & & \\
\mathsf{eval}_p(\mathcal{D}_p, \beta, \mathsf{source}(T)) & = & \mathsf{source}(\mathsf{eval}_p(\mathcal{D}_p, \beta, T)) \\
 & & \quad \text{if } \mathsf{eval}_p(\mathcal{D}_p, \beta, T) \neq \mathsf{null}, \mathsf{null} \text{ otherwise.} \\
\mathsf{eval}_p(\mathcal{D}_p, \beta, \mathsf{dest}(T)) & = & \mathsf{dest}(\mathsf{eval}_p(\mathcal{D}_p, \beta, T) \\
 & & \quad \text{if } \mathsf{eval}_p(\mathcal{D}_p, \beta, T) \neq \mathsf{null}, \mathsf{null} \text{ otherwise.} \\
\mathsf{eval}_p(\mathcal{D}_p, \beta, \mathsf{next}(X, \mathsf{changes}(x))) & = & q \text{ such that:}
\end{array}
$$

$$
\begin{array}{c}
\mathsf{time}(q) > \mathsf{time}(\mathsf{eval}_p(\mathcal{D}_p, \beta, X)) \text{ and } \mathcal{D}_p, q \vdash \mathsf{changes}(x) \text{ and there is no} \\
q' \text{ with } \mathsf{time}(\mathsf{eval}_p(\mathcal{D}_p, \beta, X)) < \mathsf{time}(q') < \mathsf{time}(q) \text{ and } \mathcal{D}_p, q' \vdash \mathsf{changes}(x) \\
\text{if such a } q \text{ exists, } \mathsf{null} \text{ otherwise.}
\end{array}
$$

$$
\mathsf{eval}_p(\mathcal{D}_p, \beta, \mathsf{next}(X, \mathsf{calls}(f))) \quad = \quad tr \text{ such that:}
$$

$$
\begin{array}{c}
(\text{if } \mathcal{D}, \mathsf{eval}_p(\mathcal{D}_p, \beta, X) \vdash \mathsf{calls}(f) \text{ then } \mathsf{time}(tr) > \mathsf{time}(\mathsf{eval}_p(\mathcal{D}_p, \beta, X)) \\
\text{otherwise } \mathsf{time}(tr) \geq \mathsf{time}(\mathsf{eval}_p(\mathcal{D}_p, \beta, X))) \\
\text{and } \mathcal{D}_p, tr \vdash \mathsf{calls}(f) \text{ and there is no} \\
tr' \text{ with } \mathsf{time}(\mathsf{eval}_p(\mathcal{D}_p, \beta, X)) < \mathsf{time}(tr') < \mathsf{time}(tr) \text{ and } \mathcal{D}_p, tr' \vdash \mathsf{calls}(f) \\
\text{if such a } tr \text{ exists, } \mathsf{null} \text{ otherwise.}
\end{array}
$$

Figure 5.4: An eval_p function for partial semantics.

We focus our attention on the eval function (see Section 3.5.3). This is responsible for taking a concrete state or transition expression, along with a dynamic run and a binding, and yielding the concrete state or transition that satisfies the expression based on the binding. In the case of the eval function, if no concrete state or transition is found given an expression, then either null or null_W is returned, depending on the type of operators found in the expression. Since the eval function deals with dynamic runs in which all information has been observed, if the relevant information is not found, then it can be assumed to not exist.

When working in the setting of partial dynamic runs, if information is not found, then it may not have been observed yet (that is, an extension of the dynamic run may contain the information needed). Hence, an eval function for this setting need not make the distinction between null and null_W, rather it can just yield null. We show in the remainder of this chapter how, despite this apparently less informative result, a semantics for partial dynamic runs can actually be more informative in cases of violation.

With this motivation for a new eval function in mind, we now introduce the eval_p function, defined in Figure 5.4. Notice that the function eval_p does not differentiate between normal and weak operators. While the original eval function would work in the partial dynamic run case, the alternative eval_p makes the definitions that we will introduce cleaner by reducing the number of cases that we must take care of. For example, the original eval function gives null_W when no information is found for a *weak* operator such as next_W. In the partial dynamic run case, this distinction has no use since we simply declare notSure if we do not have enough information to give a truth value to an atom.

$$\mathsf{partialBindings}_{\varphi}(\mathcal{D}_p, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \vdash \Gamma_n) =$$

$$\begin{cases} \bigcup_{s_1 \vdash \Gamma_1} \mathsf{partialBindings}^*_{\varphi}(\mathcal{D}_p, s_1, \forall q_2 \vdash \Gamma_2 : \cdots : \forall q_n \in \Gamma_n) & n > 1 \\ \{[q_1 \mapsto s_1] : s_1 \in \Gamma_1\} & n = 1 \end{cases}$$

$$\mathsf{partialBindings}^*_{\varphi}(\mathcal{D}_p, s_k, \forall q_{k+1} \in \Gamma_{k+1} : \cdots : \forall q_n \in \Gamma_n) =$$

$$\begin{cases} \bigcup_{s_{k+1} \vdash \Gamma_{k+1}(s_k)} \left\{ \begin{array}{l} [q_k \mapsto s_k] \dagger \beta' : \\ \quad \beta' \in \mathsf{partialBindings}^*_{\varphi}(\mathcal{D}_p, s_{k+1}, \\ \qquad \forall q_{k+2} \vdash \Gamma_{k+2} : \cdots : \forall q_n \vdash \Gamma_n) \end{array} \right\} & \Gamma_{k+1}(s_k) \neq \emptyset \\ \{[q_k \mapsto s_k]\} & \text{otherwise} \end{cases}$$

$$\mathsf{partialBindings}^*_{\varphi}(\mathcal{D}_p, s_{n-1}, \forall q_n \in \Gamma_n) =$$

$$\begin{cases} \{[q_{n-1} \mapsto s_{n-1}, q_n \mapsto s_n] : s_n \in \Gamma_n(s_{n-1})\} & \Gamma_n(s_{n-1}) \neq \emptyset \\ \{[q_{n-1} \mapsto s_{n-1}]\} & \text{otherwise} \end{cases}$$

Figure 5.5: A recursive construction for $\mathsf{partialBindings}_{\varphi}$.

5.4.3 Computing Bindings from Partial Dynamic Runs

Our existing function for computing the set of maps from variables in a CFTL specification to concrete states/transitions cannot be used for partial dynamic runs because of the fact that all quantifiers must be used before a binding is finalised. This is in line with our assumption from Section 3.6 that there must be at least one (complete) binding derived from a dynamic run with respect to a specification. However, given that our motivation for developing a partial semantics is to identify concrete states/transitions that cause failure, we need to allow evaluation of the quantifier-free part of a CFTL specification at partial bindings. For example, if we consider the CFTL specification

$$\forall s \in \mathsf{calls}(f) : \forall s' \in \mathsf{future}(s, \mathsf{calls}(g)) : \mathsf{duration}(s) < 1 \wedge \mathsf{duration}(s') < 2,$$

then the transition that represents the call to f will necessarily occur before any that represent calls to g captured by the second quantifier. Hence, if a call to f fails its constraint, then the specification cannot be satisfied and so we would like to capture this failure without having to wait for a call to g to generate a binding whose domain contains every variable.

We therefore introduce the set $\mathsf{partialBindings}_{\varphi}(\mathcal{D}_p, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ of bindings derived from the partial dynamic run $\mathcal{D}_p$ using the sequence of quantifiers from a CFTL specification. We give a recursive definition in Figure 5.5. To save space, we often omit the quantifiers (which are usually clear from context) and write $\mathsf{partialBindings}_{\varphi}(\mathcal{D}_p)$.

Some post-processing is required for this set to be used to determine the prefix of a dynamic run that first caused failure. In particular, the set of bindings used in the total semantics (Figure

3.18 on page 77) contains only bindings that are complete. If we are to allow partial bindings in our partial semantics, then we must ensure that no partial bindings are used that were never extended to form complete bindings in the total semantics (since these never contributed to the verdict given by the total semantics). To do this, letting $\mathcal{D}_p$ be a partial dynamic run that was extended to a complete dynamic run $\mathcal{D}$, we denote by

$$\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$$

the set

$$\{\beta \in \mathsf{partialBindings}_{\varphi}(\mathcal{D}_p) : \beta \sqsubset \beta' \text{ for some } \beta' \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)\}.$$

Again to save space, we often just write $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D})$ where the quantifiers are clear from context.

We now introduce a theorem that yields a corollary that we will use when we prove a correctness property of our partial semantics later. In particular, we prove that the bindings in $\mathsf{partialBindings}_{\varphi}(\mathcal{D}_p)$ are generated in the same way as those in $\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, hence that the set $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D})$ does indeed contain only bindings that will be extended to form complete bindings under a complete dynamic run.

Theorem 5. *For a CFTL specification φ, partial dynamic run $\mathcal{D}_p$ (that is a prefix of a complete dynamic run $\mathcal{D}$) and set of partial bindings $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D})$, for every $\beta \in \mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D})$, there exists some $\beta' \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ such that $\beta \sqsubset \beta'$.*

Proof. The only difference between the construction for partial bindings in Figure 5.5 and the construction for complete bindings in Figure 3.15 (page 72) is that, for the construction of the set of partial bindings, if a predicate Γ_i captures no concrete states or transitions, an empty map is given to the previous level of recursion. This is in contrast to the complete binding case in which a binding can only be generated if a concrete state/transition is found for every variable in the sequence of quantifiers (this is in fact the base case of the recursion). This policy of returning empty maps if quantifiers do not capture any concrete states/transitions is what enables partial bindings to be constructed.

Despite this, the set of partial bindings can still contain bindings that wouldn't be contained in the set of complete bindings. To see this, consider the CFTL specification with $n > 1$ quantifiers, $\forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n$ along with the partial dynamic run $\mathcal{D}_p$. We consider $n > 1$ because, for $n = 1$, the only way for a binding to not be complete is for it to be empty, in which case it wouldn't be used in monitoring. Now, without loss of generality, suppose that Γ_{n-1} captures a concrete state from $\mathcal{D}_p$. This means that (if $n > 2$) Γ_{n-2} captured a concrete state or transition (since Γ_{n-1} must depend on Γ_{n-2}), and so on until a concrete state or transition captured by Γ_1. We then construct a partial binding β that sends each q_i to the concrete state or transition that was identified by the respective Γ_i.

Consider now the extension of $\mathcal{D}_p$ to a complete dynamic run $\mathcal{D}$, but such that Γ_n does not capture any concrete state or transition given what was captured by Γ_{n-1}. Then the set

of partial bindings would include β, and there would be no complete binding β' with $\beta \sqsubset \beta'$ (recall that $\beta \sqsubset \beta'$ says that β' agrees with β on all elements in the domain of β, but also gives values to elements not in the domain of β) identified based on the complete dynamic run.

It is for this case that we include the *post-processing* step, and construct the final set of partial bindings as such:

$$\{\beta \in \mathsf{partialBindings}_{\varphi}(\mathcal{D}_p) : \beta \sqsubset \beta' \text{ for some } \beta' \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)\}.$$

This avoids the case where a partial binding is generated that would not be extended to become a complete binding. From this, we have that 1) the set of partial bindings is generated in the same way as the set of complete bindings; and 2) the post-processing step ensures that only partial bindings that will be extended to form complete bindings are included. Hence, the result follows. □

As a concrete example of the situation described in the proof, consider the CFTL specification with quantifiers

$$\forall q \in \mathsf{changes}(x) : \forall t \in \mathsf{future}(q, \mathsf{calls}(f)) : ...$$

Suppose a program contains a change of the variable x with no subsequent call of f. In this case, a partial binding would be constructed when the change of x was encountered, but no complete binding would be generated since there is no following call of f.

We now highlight that the recursive construction in Figure 5.5 can also be applied to a complete dynamic run. In this case, we then have the important corollary that the set $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}_p, \mathcal{D})$ for a complete dynamic run is the same as the set of complete bindings for that complete dynamic run. We will use this later to prove a correctness property of our partial semantics.

Corollary 2. *For a complete dynamic run $\mathcal{D}$,* $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}, \mathcal{D}) = \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$.

Proof. We show first that

$$\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}, \mathcal{D}) \subset \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n).$$

By Theorem 5, if $\mathcal{D}_p$ is set to $\mathcal{D}$, then there is no more information to observe so $\beta \sqsubset \beta'$ for $\beta \in \mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}, \mathcal{D})$ and $\beta' \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ necessarily means that $\beta = \beta'$ (since both sets $\mathsf{partialBindings}_{\varphi}(\mathcal{D}_p)$ and $\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$ are constructed in the same way in the case of a complete dynamic run, and the post-processing to construct $\mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}, \mathcal{D})$ ensures that no binding is kept which is not also found in $\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$).

We now show that

$$\mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n) \subset \mathsf{legalPartialBindings}_{\varphi}(\mathcal{D}, \mathcal{D}).$$

We argue by contradiction. Suppose that there is a binding $\beta' \in \mathsf{bindings}_{\varphi}(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots :$

$\forall_n q_n \in \Gamma_n$) but that

$$\beta' \notin \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D}).$$

However, if $\beta' \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, then for each $q_i \in \mathsf{dom}(\beta')$ (for $1 \leq i < n$) it was found that $\beta'(q_i) \in \Gamma_i(q_{i-1})$ and that $\beta'(q_1) \in \Gamma_1$. Hence, β' must also be found in $\mathsf{partialBindings}_\varphi(\mathcal{D}_p)$ by the construction of $\mathsf{partialBindings}_\varphi(\mathcal{D}_p)$. Further, if $\beta' \in \mathsf{partialBindings}_\varphi(\mathcal{D}_p)$ and $\beta' \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n)$, then $\beta' \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D})$ and we have a contradiction. Hence, it follows that

$$\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 q_1 \in \Gamma_1 : \cdots : \forall_n q_n \in \Gamma_n) \subset \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D}).$$

With containment shown in both directions, the result follows. □

Before giving our partial semantics, we will now discuss what it means for a CFTL specification to be satisfiable.

5.4.4 Unsatisfiability of CFTL Specifications

The key question that we are aiming to answer is, *for a violating dynamic run, when did violation first happen?* However, when we find that $\mathcal{D} \not\models \varphi$ for some dynamic run $\mathcal{D}$ and some CFTL specification φ, we must know whether there was a single point at which the verdict turned to false, or whether the specification was simply unsatisfiable (in which case we cannot give a single point at which violation occurred).

Our approach to determine whether a CFTL specification is unsatisfiable with respect to a dynamic run begins with determining the conditions that are necessary for a CFTL specification to be unsatisfiable. We start here because CFTL specifications only have meaning once a dynamic run is given (we will show precisely what this means later in this section), hence unsatisfiability is *with respect to* a dynamic run. Further, when we say that a CFTL specification is not unsatisfiable, hence satisfiable, this conclusion will only mean that the conditions that a CFTL specification needs to be unsatisfiable with respect to a dynamic run are not present, but the CFTL specification may still be violated.

Before beginning our discussion we remark that, from a practical point of view, software engineers do not have a habit of writing unsatisfiable specifications simply because specifications written by engineers are usually very simple. Further, many of the specifications used in our application in Chapter 7 contain only a single atom. Hence, we do not give much attention to unsatisfiability.

Unsatisfiability of CFTL specifications requires that we first have a complete dynamic run, since a dynamic run is required for most parts of a CFTL specification to have meaning. To illustrate this, we consider the specification

$$\begin{aligned} &\forall q \in \mathsf{changes}(x) : \\ &\forall c \in \mathsf{future}(q, \mathsf{calls}(f)) : \\ &\qquad \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(g))) < 3 \wedge \mathsf{duration}(\mathsf{next}(c, \mathsf{calls}(g))) > 3 \end{aligned} \tag{5.1}$$

In this case, if there is some binding identified by the quantifiers for which the next call of g along from the change of x is the same as the next call of g along from the call of f, this specification becomes unsatisfiable. However, if this is not the case then the specification is satisfiable. Hence, we can classify a CFTL specification as unsatisfiable only when we have a complete dynamic run to check. Further, we remark that this constraint poses no problems since 1) complete dynamic runs are finite and do not contain many concrete states; and 2) we will only care about determining a problematic part of the dynamic run once the entire run has been observed.

Given a CFTL specification $\varphi \equiv \forall_1 s_1 \in \Gamma_1 : \cdots : \forall_n s_n \in \Gamma_n : \psi$ and a complete dynamic run $\mathcal{D}$, we briefly describe how one can determine whether the CFTL specification is unsatisfiable. Recall that $\mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 s_1 \in \Gamma_1 : \cdots : \forall_n s_n \in \Gamma_n)$ gives the set of complete bindings derived from $\mathcal{D}$ with respect to the n quantifiers. Further we highlight that $\mathcal{D} \not\models \varphi$ if and only if there is some binding β for which $\mathcal{D}, \beta \not\models \psi$. Then, to determine whether φ is unsatisfiable, one does the following:

Expand the CFTL Specification. We construct a multi-set $\mathcal{S}$ containing an instance of ψ (the quantifier-free part of φ) for each $\beta \in \mathsf{bindings}_\varphi(\mathcal{D}, \forall_1 s_1 \in \Gamma_1 : \cdots : \forall_n s_n \in \Gamma_n)$. We denote the instance of ψ corresponding to the binding β by ψ_β. We will call each ψ_β a *formula*.

Assign Labels. The final step in our procedure for checking satisfiability of a CFTL specification with respect to a given dynamic run will be to run apply an SMT solver to a final, transformed version of the CFTL specification. To enable the SMT solver to know that two atoms with different expressions actually identify the same concrete state or transition (and therefore could be a source of unsatisfiability by applying contradicting constraints to the same quantity), we must apply labels to concrete states and transitions so that such atoms can be recognised to identify the same point in a dynamic run. Hence, to each concrete state and transition in the complete dynamic run, we assign a unique label. For concrete states, this can be the index with respect to the time-induced total order on concrete states. For transitions, this can be the pair of the indices of each of the concrete states in the transition.

Resolve Expressions. For each binding β and corresponding formula $\psi_\beta \in \mathcal{S}$, we iterate through all concrete states and transitions. We consider concrete states; for each one, we replace any expressions in ψ_β that refer to that concrete state with the concrete state's label. We do the same for transitions. For example,

$$\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(g))) < 3 \wedge \mathsf{duration}(\mathsf{next}(c, \mathsf{calls}(g))) > 3$$

would become

$$\mathsf{duration}(\langle l_1, l_2 \rangle) < 3 \wedge \mathsf{duration}(\langle l_1, l_2 \rangle) > 3$$

if $\mathsf{next}(q, \mathsf{calls}(f))$ and $\mathsf{next}(c, \mathsf{calls}(g))$ were evaluated to the transition with label $\langle l_1, l_2 \rangle$ given the relevant dynamic run.

We denote the formula ψ_β after this transformation by $\mathsf{labelled}(\psi_\beta)$. Significantly, $\mathsf{labelled}(\psi_\beta)$ has no ambiguity with respect to which atoms constrain which quantities because, if there were

different expressions being used to obtain the same quantities in the dynamic run, they have been assigned the same labels. Finally, since each expression/atom can be given only one label, we define the atoms of $\mathsf{labelled}(\psi_\beta)$, denoted by $A_{\mathsf{labelled}(\psi_\beta)}$, as the parts of the formula obtained by replacing the corresponding atoms in ψ_β with their unique labels.

Translate to a Satisfiability Modulo Theories (SMT) Instance. For each binding β, derived from $\mathcal{D}$, we translate $\mathsf{labelled}(\psi_\beta)$ into a SMT instance that has variables and predicates, with no quantifiers. We denote the translation by $\mathsf{SMT}(\mathsf{labelled}(\psi_\beta))$.

If this instance is satisfiable, ψ_β is satisfiable given the relevant dynamic run and we move onto the next binding. Otherwise, we have found a ψ_β which, given a particular dynamic run, cannot be satisfied and so $\mathcal{D} \not\models \varphi$.

Determine Unsatisfiability. Each formula ψ_β is unsatisfiable if and only if $\mathsf{SMT}(\mathsf{labelled}(\psi_\beta))$ is unsatisfiable. To prove unsatisfiability of a CFTL specification over a dynamic run, it suffices to show the existence of a single binding β derived from the dynamic run for which $\mathsf{SMT}(\mathsf{labelled}(\psi_\beta))$ is unsatisfiable.

We take as an example the specification given above

$$\begin{aligned} &\forall q \in \mathsf{changes}(x) : \\ &\forall c \in \mathsf{future}(q, \mathsf{calls}(f)) : \\ &\qquad \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(g))) < 3 \wedge \mathsf{duration}(\mathsf{next}(c, \mathsf{calls}(g))) > 3. \end{aligned} \tag{5.2}$$

If the dynamic run over which we evaluate this specification gives the same labels to $\mathsf{next}(q, \mathsf{calls}(f))$ and $\mathsf{next}(c, \mathsf{calls}(g))$ (because they refer to the same transition), then this specification is unsatisfiable because the same quantity is required to be less than 3 and greater than 3 at the same time. However, if $\mathsf{next}(q, \mathsf{calls}(f))$ and $\mathsf{next}(c, \mathsf{calls}(g))$ are not given the same labels (because they refer to different transitions), then this specification is not unsatisfiable (hence, satisfiable). However, it can still be violated if, for example, $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) = 5$, which clearly violates the constraint placed by the specification.

We do not include further discussion, since unsatisfiability of CFTL specifications is not a problem we face in practice.

5.4.5 The Semantics Function

We now present a partial semantics for (satisfiable) CFTL specifications in Figure 5.6. The structure of the partial semantics necessarily differs greatly from that of the total semantics. This is because we don't have a 2-valued truth domain, and so cannot simply use a semantics relation to encode the two possible values (the total semantics relation encodes true and false via containment in the relation). With this in mind, we must use a *semantics function* [BLS11] which takes as input either a partial or complete dynamic run, a binding and the quantifier-free part of a CFTL specification. We highlight that the semantics function is applicable to both partial and complete dynamic runs, a fact that will be of use later in the section.

Since the semantics function requires the quantifier-free part of a CFTL specification, we denote the verdict after checking a (partial or complete) dynamic run $\mathcal{D}_p$ for satisfaction of a (satisfiable) CFTL specification by

$$[\mathcal{D}_p, \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \phi] = \prod_{\beta \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}_p, \mathcal{D})} [\mathcal{D}_p, \beta, \phi].$$

We reiterate that the partial semantics is a mechanism to be used to determine the concrete state in a dynamic run at which the verdict becomes false. We use this mechanism only when we have already processed the dynamic run (hence, have obtained a complete dynamic run) and it has given a false verdict under the total semantics. Now, in order to use this partial semantics to determine where a verdict becomes false in a dynamic run, we use the following procedure:

For a complete dynamic run $\mathcal{D}$, if $\mathcal{D} \not\models \varphi$ for φ is satisfiable, then we have two possibilities:

- Not everything that was needed was observed, and the operators used in φ made the verdict default to false.

  We call this "false because not enough was observed".

- A constraint was genuinely violated. In this case, there are two possible situations for whether enough data was observed. The first is that not everything that was needed was observed, but the operators used in φ did not make the verdict default to false. The second is that everything was observed, so the types of the operators used did not matter.

  We call this situation "genuinely false".

We check if the verdict was "false because not enough was observed" by looking at the verdict generated by

$$\prod_{\beta \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}_p, \mathcal{D})} [\mathcal{D}_p, \beta, \phi].$$

If this verdict is notSure, then we have "false because not enough was observed", meaning that the false verdict was reached by the total semantics for the same reason.

Alternatively, if the partial semantics gives false, then the specification was violated because a value observed in the dynamic run violated a constraint. In this case, we have Theorem 6, which shows that false from the partial semantics implies false from the total semantics.

We remark that we cannot show the result in the other direction because, as already described, if the total semantics yields false this can be either because not enough information was observed or a value genuinely violated a constraint. The lack of a result in the opposite direction does not pose a problem; we only need to be able to define the type of violation yielded by the total semantics, and the result below enables us to do this via the partial semantics.

Theorem 6. *For a complete dynamic run $\mathcal{D}$, if $\prod_{\beta \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D})} [\mathcal{D}, \beta, \phi] = \mathsf{false}$, then $\mathcal{D} \not\models \varphi$ because a constraint was violated.*

Proof. If $\prod_{\beta \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D})} [\mathcal{D}_p, \beta, \phi] = \mathsf{false}$ then there is a binding

$$\beta \in \mathsf{legalPartialBindings}_\varphi(\mathcal{D}, \mathcal{D})$$

for which $[\mathcal{D}, \beta, \phi] = \mathsf{false}$. If this is the case, then we must go through cases in the partial semantics in Figure 5.6:

- If we have $[\mathcal{D}, \beta, \phi_1 \vee \phi_2] = [\mathcal{D}, \beta, \phi_1] \sqcup [\mathcal{D}, \beta, \phi_2] = \mathsf{false}$ then, by the ordering on the truth domain, we must have $[\mathcal{D}, \beta, \phi_1] = \mathsf{false}$ and $[\mathcal{D}, \beta, \phi_2] = \mathsf{false}$. To see the consequences of this, we must consider the other cases.
- We take the part of the semantics function $[\mathcal{D}, \beta, S(x) = v]$ and equate it to false. In doing this, we have

$$[\mathcal{D}, \beta, S(x) = v] = \mathsf{false} \text{ iff } (\mathsf{eval}_p(\mathcal{D}, \beta, S) \neq \mathsf{null} \ \wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S)(x) \neq v)$$

  then we have $\mathsf{eval}_p(\mathcal{D}, \beta, S) \neq \mathsf{null} \ \wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S)(x) \neq v$, meaning that the value required was found, but it violated the constraint present in the specification.
- We take the part of the semantics function $[\mathcal{D}, \beta, S_1(x_1) = S_2(x_2)]$ and equate it to false. In doing this, we have

$$\begin{aligned}[\mathcal{D}, \beta, S_1(x_1) = S_2(x_2)] = \ \mathsf{false} \text{ iff } (&\mathsf{eval}_p(\mathcal{D}, \beta, S_1) \neq \mathsf{null} \\ &\wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S_2) \neq \mathsf{null} \\ &\wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S_1)(x_1) \neq \mathsf{eval}_p(\mathcal{D}, \beta, S_2)(x_2))\end{aligned}$$

  then we have $\mathsf{eval}_p(\mathcal{D}, \beta, S_1) \neq \mathsf{null} \ \wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S_2) \neq \mathsf{null} \ \wedge \ \mathsf{eval}_p(\mathcal{D}, \beta, S_1)(x_1) \neq \mathsf{eval}_p(\mathcal{D}, \beta, S_2)(x_2)$, meaning that the values required were found, but they violated the constraint present in the specification.

In each case, we know that the binding from the partial semantics is reflected in the total semantics because of Corollary 2.

We will not consider all cases since the arguments are the same. However, by looking at the individual cases present in the partial semantics, we see that a false verdict means also that $\mathcal{D} \not\models \varphi$, since the conditions that are true for $[\mathcal{D}, \beta, \phi] = \mathsf{false}$ are precisely those required for $\mathcal{D}, \beta \not\models \varphi$. The result follows, since the violation according to the total semantics that is implied must be because of a constraint violation, and not insufficient information being found (we have assumed that the partial semantics has given false, and not $\mathsf{notSure}$). □

This result describes the relationship between the partial semantics and total semantics that we need to define at which point a dynamic run violated a specification (if the violation was due to a constraint being violated).

5.4.6 Concrete States that are Falsifying

Using our partial semantics, we can define what it means to say that a concrete state from a dynamic run is *falsifying* of a (satisfiable) CFTL specification. Given a total dynamic run $\mathcal{D}$ let $\mathcal{D}_p(q)$ be the partial dynamic run that is the prefix of $\mathcal{D}$ ending with concrete state q. For a CFTL specification φ such that $\mathcal{D} \not\models \varphi$, the concrete state q is falsifying if and only if:

1. $[\mathcal{D}_p(q), \varphi] = \mathsf{false}$.

$$
\begin{array}{lcl}
[\mathcal{D}_p, \beta, \mathit{true}] & = & \mathsf{trueSoFar} \\
[\mathcal{D}_p, \beta, \phi_1 \vee \phi_2] & = & [\mathcal{D}_p, \beta, \phi_1] \sqcup [\mathcal{D}_p, \beta, \phi_2] \\
[\mathcal{D}_p, \beta, \neg\phi] & = & \neg[\mathcal{D}_p, \beta, \phi] \\
[\mathcal{D}_p, \beta, S(x) = v] & = & \begin{cases}
\mathsf{trueSoFar} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S)(x) = v) \\
\mathsf{notSure} & \text{iff } \mathsf{eval}_p(\mathcal{D}_p, \beta, S) = \mathsf{null} \\
\mathsf{false} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S)(x) \neq v)
\end{cases} \\
[\mathcal{D}_p, \beta, S_1(x_1) = S_2(x_2)] & = & \begin{cases}
\mathsf{trueSoFar} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S_1) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S_2) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S_1)(x_1) = \mathsf{eval}_p(\mathcal{D}_p, \beta, S_2)(x_2))) \\
\mathsf{notSure} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S_1) = \mathsf{null} \\
 & \quad \vee\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S_2) = \mathsf{null}) \\
\mathsf{false} & \text{iff } \mathsf{eval}_p(\mathcal{D}_p, \beta, S_1) \neq \mathsf{null} \\
. & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S_2) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S_1)(x_1) \neq \mathsf{eval}_p(\mathcal{D}_p, \beta, S_2)(x_2))
\end{cases} \\
[\mathcal{D}_p, \beta, S(x) \in [n, m]] & = & \begin{cases}
\mathsf{trueSoFar} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S)(x) \in [n, m]) \\
\mathsf{notSure} & \text{iff } \mathsf{eval}_p(\mathcal{D}_p, \beta, S) = \mathsf{null} \\
\mathsf{false} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, S) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{eval}_p(\mathcal{D}_p, \beta, S)(x) \notin [n, m])
\end{cases} \\
[\mathcal{D}_p, \beta, \mathsf{duration}(T) \in (n, m)] & = & \begin{cases}
\mathsf{trueSoFar} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, T) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{duration}(\mathsf{eval}_p(\mathcal{D}_p, \beta, T)) \in (n, m)) \\
\mathsf{notSure} & \text{iff } \mathsf{eval}_p(\mathcal{D}_p, \beta, T) = \mathsf{null} \\
\mathsf{false} & \text{iff } (\mathsf{eval}_p(\mathcal{D}_p, \beta, T) \neq \mathsf{null} \\
 & \quad \wedge\ \mathsf{duration}(\mathsf{eval}_p(\mathcal{D}_p, \beta, T)) \notin (n, m))
\end{cases}
\end{array}
$$

Figure 5.6: The 3-valued semantics function [.].

2. Given the previous state q' in $\mathcal{D}$ (if it exists), $[\mathcal{D}_p(q'), \varphi] = \mathsf{trueSoFar}$ or $[\mathcal{D}_p(q'), \varphi] = \mathsf{notSure}$.

The falsifying concrete state of a dynamic run is necessarily unique by impartiality. That is, the only way in which there could be multiple falsifying concrete states would be if the verdict could change from false, ready to change back again later. Impartiality prevents this.

We remark that, given a CFTL specification φ and a dynamic run $\mathcal{D}$ for which $\mathcal{D} \not\models \varphi$ with φ being satisfiable with respect to $\mathcal{D}$, one can identify the falsifying concrete state in $\mathcal{D}$ by considering progressively longer prefixes of $\mathcal{D}$.

Notice that we do not discuss transitions that are falsifying. This is because all quantities around a transition that can affect the verdict are at its beginning or end, and in either case it suffices to discuss a concrete state.

5.5 Paths with Falsifying Concrete States

We will now link the notion of a program path with a concrete state that is attained during a dynamic run. In this case, the path we consider may no longer be complete (it may not end in a symbolic state that is final).

Given a branch-aware dynamic run $\mathcal{D}$ and a CFTL specification φ such that $\mathcal{D} \not\models \varphi$, there is necessarily a partial dynamic run $\mathcal{D}_p$ that is a prefix of $\mathcal{D}$ such that $[\mathcal{D}_p, \varphi] = \mathsf{false}$ and whose last concrete state is the falsifying concrete state in $\mathcal{D}$. This prefix of a branch-aware dynamic run is necessarily unique, since the falsifying concrete state is unique.

If $\mathcal{D}_p$ is a partial version of $\mathcal{D}$, it is also branch-aware since the only way to destroy branch-awareness is to remove concrete states that are not the final one (removing the final one just results in reconstruction of a shorter path) and taking prefixes can be seen as iteratively removing the final concrete state of a series of partial dynamic runs. This means that by considering the partial dynamic run that ends with a falsifying concrete state, we can still reconstruct the path through the symbolic control-flow graph.

With this in mind, fixing some dynamic run $\mathcal{D}$ and symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$, we denote by $\pi(s)$ the path reconstructed through the symbolic control-flow graph up to the symbolic state associated with the concrete state s in $\mathcal{D}$. This path, reconstructed using Algorithm 8, will include information only obtained during a program run such as the number of iterations completed by each loop. Notice that in this notation we use the concrete state s and not its symbolic state; this is to enforce that the path should require information from a program run, rather than just being the shortest path from v_s to the concrete state's symbolic state.

5.6 Operations over Paths

We now introduce operations that one can perform over paths, including *subtraction* and *intersection*. In particular, for path intersection, we will give a criterion to constrain sets of paths over which one can perform intersection. We will then give a formal definition and an algorithm for computing the intersection of multiple paths. Being able to perform these operations

over paths will allow us to derive information from sets of paths reconstructed from multiple dynamic runs.

5.6.1 Path Subtraction

The first operation that we describe is a straightforward one; given a dynamic run $\mathcal{D}$ over a symbolic control-flow graph $\mathsf{SCFG}(P)$ with two concrete states s, s' such that $\mathsf{time}(s) < \mathsf{time}(s')$, we would like to construct the path *from s to s'* in the $\mathsf{SCFG}(P)$. When we refer to the path between two concrete states, we mean the path between their symbolic states, but taking into account the information present in the dynamic run such as loop iterations.

It is necessary to first reconstruct both $\pi(s)$ and $\pi(s')$ since there could be loops between s and s', but this could be easily made more efficient by starting from the path $\pi(s)$ and continuing reconstruction from there since $\mathsf{time}(s) < \mathsf{time}(s') \implies \pi(s) \sqsubset \pi(s')$ within the same dynamic run.

Once the paths $\pi(s)$ and $\pi(s')$ have been computed, we denote by $\pi(s') - \pi(s)$ the path between them and define it, for concrete states s and s' with $\mathsf{time}(s) < \mathsf{time}(s')$, by

$$\pi(s') - \pi(s) = \langle \sigma_u, \sigma_{u+1} \rangle \dots \langle \sigma_{v-1}, \sigma_v \rangle$$

where $\pi(s) = \langle \sigma_1, \sigma_2 \rangle \dots \langle \sigma_{u-1}, \sigma_u \rangle$ and $\pi(s') = \langle \sigma_1, \sigma_2 \rangle \dots \langle \sigma_{v-1}, \sigma_v \rangle$.

5.6.2 Path Intersection

Intersecting paths is a procedure that tells us at which points the paths diverge. This is more involved than the straightforward path subtraction and requires the introduction of a new technique for comparison of multiple program paths.

We begin by fixing a symbolic control-flow graph $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$ a symbolic state $\sigma \in V$ and a family of branch-aware dynamic runs $\mathcal{D}_1, \dots, \mathcal{D}_k$ over $\mathsf{SCFG}(P)$ such that each $\mathcal{D}_i$ contains a single concrete state that corresponds to the symbolic state σ in V. We will first consider paths leading to this σ, reconstructed with respect to the relevant concrete state in each dynamic run, and how we can compare them. Eventually, we will consider paths between pairs of symbolic states.

For s_i from $\mathcal{D}_i$ such that the symbolic state with which s_i is associated is the σ to which we refer above, we already know how to reconstruct the path $\pi(s_i)$. It is therefore straightforward to obtain the family of paths $\pi(s_1), \dots, \pi(s_k)$ from the family of branch-aware dynamic runs.

We will now turn our attention to determining what these paths have in common. In order to do this, we will determine how to compare this family of paths so that we can identify a set of paths through $\mathsf{SCFG}(P)$ that each form a *section* of each path $\pi(s_i)$. We say that a path $\pi = e_1 \dots e_n$ is a *section* of $\pi' = e'_1 \dots e'_m$ if there is $1 \leq p \leq m-n$ such that, for every $1 \leq j \leq n$, $e_j = e'_{p+j}$. These sections of paths will be the subpaths taken by each path in the family in a certain *region of disagreement*. For example, if we determine that all paths disagree on a certain conditional block, then we will determine that all paths have a *region of disagreement* at this conditional block, and therefore each path will have a section that corresponds to the path it took through this block. In order to determine these regions of disagreement for sets of

paths through symbolic control-flow graphs, we now introduce our approach that is based on context-free grammars.

Context-Free Grammars for Representing Paths

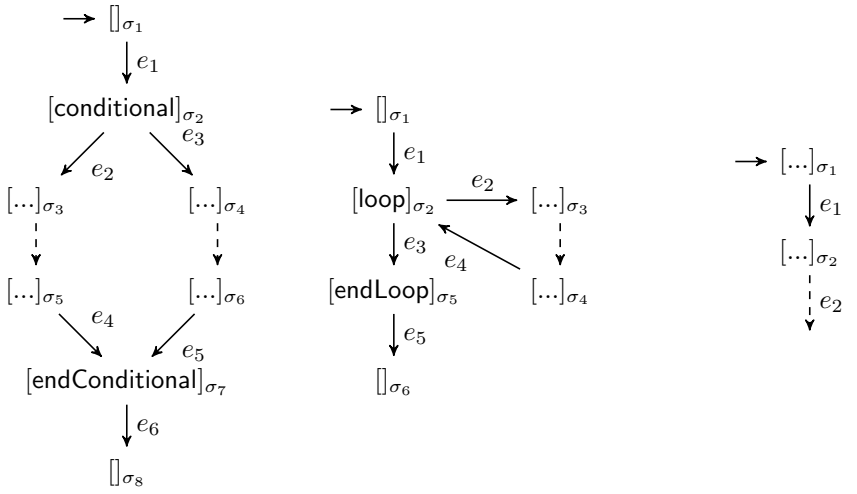
We need to develop a representation of paths that allows us to perform comparison easily; the existing, sequential form makes it difficult to decide which parts of paths correspond to parts of other paths, especially when loops are involved. Our solution is to recognise that one can derive an LL(1) context-free grammar from a symbolic control-flow graph and represent paths as being generated by such a grammar.

We remark that one could choose to convert a symbolic control-flow graph into a finite state automaton and use existing methods to generate a context-free grammar from the finite state automaton. One might then think of using regular expressions instead of context-free grammars. In particular, despite the fact that a path through a program has certain constraints, including that every construct must be closed (for example, all conditionals and loops must be exited), making only final states of the automaton accepting would implicitly enforce these constraints in a language that would still be regular. Hence, the language of paths through a symbolic control-flow graph can be captured by a regular expression, but this does not help with comparison because we do not get a structure that highlights differences in branching between multiple paths. For this reason, we use context-free grammars and, rather than deriving them from symbolic control-flow graphs by first converting to a finite state automaton, we introduce a new approach.

Symbolic Control-Flow Graphs to Context-Free Grammars

We give a subset of our full schema for deriving a context-free grammar from a symbolic control-flow graph in Figure 5.7. We omit the case for while-loops because of its similarity to the case for for-loops. For a symbolic control-flow graph, the context-free grammar that we construct has symbolic states as non-terminal symbols and edges as terminal symbols; a path is a sequence of edges, so this is a natural choice for non-terminal symbols. One can use this schema to construct a context-free grammar by recursively applying the rules from the inner-most structures of the symbolic control-flow graph, working towards the outer ones. For example, if a symbolic control-flow graph encodes two nested loops, then the grammar rules are first generated for the inner loop, and are then used in the grammar rules of the outer loop. An example result of this procedure is given in Figure 5.8, which demonstrates application of this recursive strategy to a simple program with a single loop.

The novelty in our approach is seen in the structure of the context-free grammars generated by the schema. In particular, by looking at the rule for $\vec{\sigma_1}$ generated for the loop in the middle in Figure 5.7, one can see that the body of the loop and the post-loop edges are generated by separate rules. This gives rise to parse trees of the form shown in Figure 5.9. In this parse tree, it is clear that the body of the loop is represented as its own subtree of the complete parse tree. Characteristics like this make our intersection approach useful for determining differences in paths taken through constructs such as conditionals and loops.



$$
\begin{array}{lcl}
\vec{\sigma_1} & \rightarrow & e_1\ \vec{\sigma_2}\ \vec{\sigma_7} \\
\vec{\sigma_2} & \rightarrow & e_2\ \vec{\sigma_3} \mid e_3\ \vec{\sigma_4} \\
\vec{\sigma_3} & \rightarrow & \ldots \\
\vec{\sigma_4} & \rightarrow & \ldots \\
 & \vdots & \\
\vec{\sigma_5} & \rightarrow & e_4 \\
\vec{\sigma_6} & \rightarrow & e_5 \\
\vec{\sigma_7} & \rightarrow & e_6\ \vec{\sigma_8}
\end{array}
\qquad
\begin{array}{lcl}
\vec{\sigma_1} & \rightarrow & e_1\ \vec{\sigma_2}\ \vec{\sigma_5} \\
\vec{\sigma_2} & \rightarrow & e_2\ \vec{\sigma_3} \mid e_3 \\
\vec{\sigma_3} & \rightarrow & \ldots \\
 & \vdots & \\
\vec{\sigma_5} & \rightarrow & e_5\ \vec{\sigma_6}
\end{array}
\qquad
\begin{array}{lcl}
\vec{\sigma_1} & \rightarrow & e_1\ \vec{\sigma_2} \\
\vec{\sigma_2} & \rightarrow & \ldots
\end{array}
$$

Figure 5.7: A subset of the full schema for deriving a context-free grammar from a symbolic control-flow graph.

We now prove an important result that will allow a more efficient parse tree derivation algorithm.

Theorem 7. *A context-free grammar C derived from the schema in Figure 5.7 is LL(1).*

Proof. We show that the grammar generated by each case of the schema in Figure 5.7 is LL(1), that is, paths generated by the grammars can be reconstructed by using the next edge in the path prefix generated so far to determine which rule to follow to generate the remainder of the path.

In each case, the terminal symbols (edges e_i) appear in the generated grammar only once, and the first symbol in a string generated by a rule is always a terminal symbol. Hence given any path prefix there can be only one rule to choose to generate the next part of the path.

We show this explicitly by giving the parsing table for each grammar. For the conditional

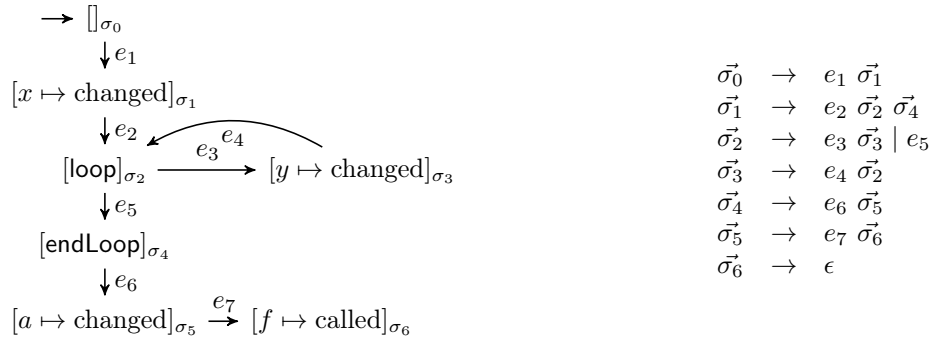


Figure 5.8: A symbolic control-flow graph and its context-free grammar.

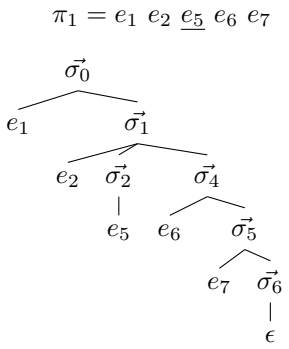


Figure 5.9: An example parse tree of a path through a symbolic control-flow graph.

case, we have the parsing table:

	e_1	e_2	e_3	e_4	e_5	e_6
$\vec{\sigma_1}$	$e_1\ \vec{\sigma_2}\ \vec{\sigma_7}$	–	–	–	–	–
$\vec{\sigma_2}$	–	$e_2\ \vec{\sigma_3}$	$e_3\ \vec{\sigma_4}$	–	–	–
$\vec{\sigma_3}$	–	–	–	–	–	–
$\vec{\sigma_4}$	–	–	–	–	–	–
$\vec{\sigma_5}$	–	–	–	e_4	–	–
$\vec{\sigma_6}$	–	–	–	–	e_5	–
$\vec{\sigma_7}$	–	–	–	–	–	$e_6\ \vec{\sigma_8}$

For the for-loop case we have the parsing table:

	e_1	e_2	e_3	e_4	e_5	e_6
$\vec{\sigma_1}$	$e_1\ \vec{\sigma_2}\ \vec{\sigma_5}$	–	–	–	–	–
$\vec{\sigma_2}$	–	$e_2\ \vec{\sigma_3}$	e_3	–	–	–
$\vec{\sigma_3}$	–	–	–	–	–	–
$\vec{\sigma_4}$	–	–	–	–	–	–
$\vec{\sigma_5}$	–	–	–	–	$e_5\ \vec{\sigma_6}$	–

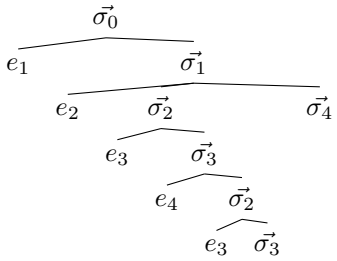


Figure 5.10: A partial parse tree.

We omit the while-loop case because of its similarity to the for-loop case, and we omit the single statement case because it is clearly LL(1). Now, the parsing tables above show that, for each non-terminal symbol, there is only one rule that can be used to generate each potential next symbol when parsing a path based on the grammars generated by the schema show in Figure 5.7.

Since the grammars generated by each case in the schema are LL(1), and the schema is applied to subprograms represented in a symbolic control-flow graph, the overall grammar generated from the entire symbolic control-flow graph is LL(1). □

Given the LL(1) context-free grammars generated by the schema in Figure 5.7, parse trees can be constructed trivially by iterating from left to right over the non-terminal symbols generated so far, with a lookahead of one symbol to decide which case should be used to substitute for each non-terminal symbol.

To finish this introduction of parse trees for paths through symbolic control-flow graphs, we introduce our notation for such trees. We denote by $T_C(\pi)$ the parse tree of a path π derived with respect to the context-free grammar C. However, we usually write $T(\pi)$ if C is clear from context.

Partial Parse Trees

If we consider the path $\pi(s)$ leading to some concrete state s, then we have the problem that this path will usually not be complete in the symbolic control-flow graph over which we reconstructed it. This is the case because concrete states can be associated with symbolic states that are not final. Hence, we must cater for this in our construction of parse trees. Figure 5.10 gives an example of a *partial parse tree*, that is, a parse tree in which some leaves are non-terminal symbols.

Definition 6. *The* partial parse tree *of a path $e_1 \dots e_k$ (such that $e_k = \langle \sigma, \sigma' \rangle$ with σ' not final) through a symbolic control-flow graph with respect to a context-free grammar is a parse tree for which the* leaves *may be terminal and non-terminal symbols from the grammar.*

We add the constraint that these partial parse trees that represent paths up to concrete states can only have leaves with non-terminal symbols to the right of all of the leaves with terminal symbols (ie, edges). Later we will see parse trees in which this constraint does not hold.

$$\mathsf{intersect}(l, l') = \begin{cases} \mathsf{null} & \text{if } l \neq l' \\ l & \text{otherwise} \end{cases}$$

$$\mathsf{intersect}(\langle r, [h_1, \ldots, h_n]\rangle, \langle r', [h'_1, \ldots, h'_m]\rangle) = \begin{cases} \langle r, [\ldots \mathsf{intersect}(h_i, h'_i) \ldots]\rangle & \text{if } r = r' \wedge n = m \\ r & \text{if } r = r' \wedge n \neq m \\ \mathsf{null} & \text{if } r \neq r' \end{cases}$$

Figure 5.11: A recursive definition of parse tree intersection.

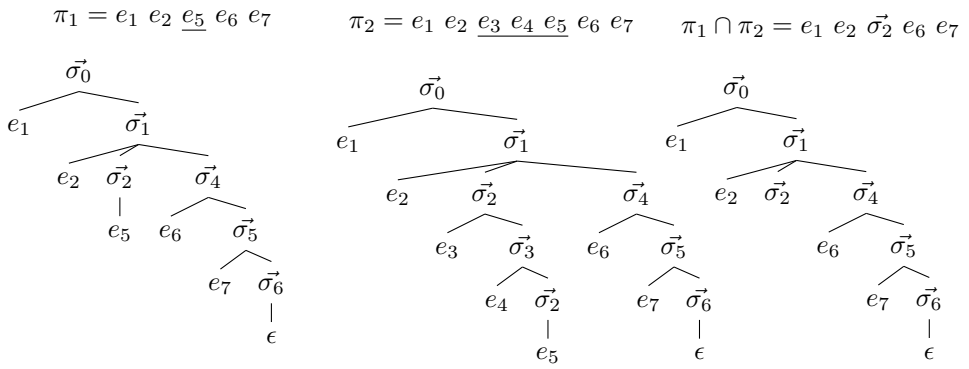


Figure 5.12: The parse trees of two paths, and their intersection.

Intersecting Parse Trees

Our intersection approach considers two paths $\pi(s), \pi(s')$ and requires the construction of $T(\pi(s))$ and $T(\pi(s'))$. Figure 5.11 gives a recursive definition of the intersection, which we denote by $T(\pi(s)) \cap T(\pi(s'))$ (or, to abuse notation, $\pi(s) \cap \pi(s')$). The result of intersection is therefore a new parse tree which contains only subtrees that are present in both parse trees used, and at the same location. An example of intersection over two paths through a loop is given in Figure 5.12. Both paths traversed the loop a different number of times, and this difference is reflected in the deletion of the subtree that represents the traversal of the loop. Algorithm 9 computes the intersection of a family of path parse trees based on the definition in Figure 5.11 (which is only in terms of two path parse trees). It is important to notice here that Algorithm 9 given a family $T(\pi(s_1)), \ldots, T(\pi(s_n))$ of path parse trees *builds* the intersection by processing all trees and then constructing a final result, rather than computing $T(\pi(s_1)) \cap T(\pi(s_2))$ and then intersecting the result with $T(\pi(s_3))$, and so on. The result of this *processing and then building* approach to intersection is denoted by $T(\pi(s_1)) \cap \cdots \cap T(\pi(s_n))$. We will now describe the intuition of the algorithm.

The approach is based on the fact that a tree can be uniquely described by the set of paths from its root vertex to its leaves and, as a result, the intersection of multiple trees can be defined by an iterative refinement of this set of paths across all trees. Hence, we start with the complete set of paths from the root of $T(\pi_1)$ to its leaves and then progressively trim these paths until we have a final set of paths from which we can construct the resulting intersection.

Algorithm 9 Construct the intersection parse tree of a family of path parse trees $T(\pi_1), \ldots, T(\pi_n)$.

```
1: Π ← {π′ : π′ is a sequence of vertices from the root of T(π_1) to a leaf}
2: for T(π_i) ∈ {T(π_2), ..., T(π_n)} do
3:   for π′_i ∈ Π do
4:     curr ← root of T(π_i)
5:     trimIndex ← 0
6:     for v ∈ π_i with index i_v do        ▷ Follow the path π_i as far down T(π_i) as possible
7:       if i_v + 1 < |π′_i| then           ▷ Check that we can still follow the path
8:         found ← false
9:         for child c of curr do           ▷ Try to find the child vertex that we can follow
10:          if v′ = c for i_v′ = i_v + 1 then
11:            curr ← c
12:            found ← true                 ▷ If we found it, acknowledge it
13:        if ¬ found then
14:          trimIndex ← i_v + 1            ▷ If we couldn't follow the path all the way, trim it
15:          break
16:      else
17:        trimIndex ← |π′_i|               ▷ If we couldn't follow the path all the way, trim it
18:    π_i ← ⟨v_1, ..., v_k⟩ such that k = trimIndex
19: return new parse tree based on modified Π   ▷ Return new parse tree from modified Π
```

Π is initialised as the set of all paths from the root of $T(\pi_1)$ to its leaves; this set will be progressively refined. We then iterate through the family of parse trees, ignoring the first, and perform the refinement procedure based on each parse tree.

Refinement is performed by iterating through the current set Π and attempting to follow each path until we reach disagreement (either a mismatch of subtrees, or the lack of a subtree). Whenever we reach disagreement, we record the depth in the tree to which we have traversed and use that to trim the current path in Π.

By performing this refinement on all paths across all parse trees, the result is a set of paths through parse trees that traverse only the vertices that exist in all parse trees in the intersection.

With this intuition, we argue that processing all parse trees and then building the intersection at the end is more efficient than computing $T(\pi(s_1)) \cap T(\pi(s_2))$ and then intersecting the result with $T(\pi(s_3))$ (and so on) because the first version involves computing a set of paths through a parse tree and refining them for each of the $n-1$ other parse trees. The second version would correspond to $n-1$ calls of the algorithm, which would mean that reconstruction of a new intersection parse tree based on the set of refined paths would be performed $n-1$ times, rather than just once.

Finally, this intersection algorithm's worst case performance is triggered if either of the following is true:

- All parse trees were identical (then no refinement would ever take place, and the result would be traversal of every path through every parse tree, a possibly very slow process).

  This could be countered by serialising the parse trees and checking for equivalence of the serialised forms.

- The parse trees are given in an order such that larger parse trees are found earlier, and smaller parse trees are found towards the end of the list. The result in this case would be traversing many longer paths through many parse trees, and only determining towards the end of the list that these paths may not be needed.

  This could be countered by ordering the parse trees according to size. While a simple measure of size does not encode more complex characteristics of the structure of the parse trees, this would not affect our algorithm much because all parse trees are derived from paths through the same symbolic control-flow graph. Hence, the structure would be similar, so a numerical measure of size would suffice.

5.6.3 Working with Path Parameters

Consider the intersection $T(\pi(s_1)) \cap \cdots \cap T(\pi(s_n))$ computed by Algorithm 9, of which an example is given in Figure 5.12. The right-most parse tree, the result of the intersection of the two on the left, has a leaf which is a non-terminal symbol. By construction, this means that, if we were to follow the path to that leaf through each of the parse trees used to compute the intersection, we would not find the same subtree in all parse trees. Multiple parse trees could have the same subtree in the position indicated by the path to the leaf but, if there is a single one that does not, this subtree will not be included in the intersection.

We now consider the path obtained from an intersection parse tree, denoted by $\mathsf{path}(T(\pi(s_1)) \cap \cdots \cap T(\pi(s_n)))$, that is obtained by reading the leaves. We refer specifically to *intersection parse trees* because the result of reading the leaves of such parse trees is still a representation of the intersection, but in the form of a sequence rather than a tree. If $\mathsf{path}(T(\pi(s_1)) \cap \cdots \cap T(\pi(s_n)))$ were to contain a non-terminal symbol $\vec{\sigma}$, this would indicate that the rule $\vec{\sigma}$ in the context-free grammar used to derive the parse tree can be used to *fill in* the gap in the path created by the non-terminal symbol. Hence, we refer to each non-terminal symbol $\vec{\sigma}$ (corresponding to the symbolic state σ) that is a leaf as a *path parameter*.

We call a path a *parametric path* if there are indeed path parameters present. For example, taking the intersection parse tree in Figure 5.12, the resulting path is

$$e_1 \; e_2 \; \vec{\sigma_2} \; e_6 \; e_7$$

with the path parameter $\vec{\sigma_2}$ to which we can give values obtained by using the rule $\vec{\sigma_2}$ from the context-free grammar.

Now, giving the symbolic state or non-terminal symbol does not always suffice to uniquely identify a path parameter. One must either give the pair $\langle i, \vec{\sigma} \rangle$ to denote the i^{th} occurrence of $\vec{\sigma}$ starting from the beginning of the path, or a path π_T from the root of the parse tree to the node at which the $\vec{\sigma}$ to which we refer is found. Using one of these unique identifiers for a path parameter, it can be given a *value* in the form of a path generated by using the rule $\vec{\sigma}$ from the context-free grammar. For the intersection $I = T(\pi(s_1)) \cap \cdots \cap T(\pi(s_n))$, we denote by

$$\mathsf{path}(I) \downarrow [\langle i, \vec{\sigma} \rangle \mapsto \pi] \text{ or } \mathsf{path}(I) \downarrow [\pi_T \mapsto \pi],$$

with the path π_T leading to the path parameter $\langle i, \vec{\sigma} \rangle$, the path obtained by substituting π in the relevant place. Here, it must be possible to generate π using the rule $\vec{\sigma}$ from the relevant context-free grammar. Further, it is possible that the path resulting from the substitution is still parametric if there were multiple parameters (and we only gave a value for one of them).

Finally, it is straightforward to determine the value that some path π used in an intersection gives to a path parameter by 1) constructing $T(\pi)$ and 2) taking the appropriate π_T, following it through $T(\pi)$ and remembering the subtree found. The path π_T is guaranteed to be found in each parse tree used in the intersection by the path refinement approach used in the intersection algorithm.

We remark that all subtrees found by following paths π_T must be able to generate new paths because the only way a node can have children in a parse tree is if that node is a non-terminal symbol and can generate a new sequence of symbols.

5.6.4 Describing Path Disagreement

Now that we have the machinery to compare paths and determine specifically how they differ, we can address our initial problem of determining how paths behave in a region of disagreement. We recall that a *region of disagreement* of a family of paths is a region in the symbolic control-flow graph (a conditional block, a for-loop, etc) through which paths in the family took a different subpath. The key idea encoded in a *region of disagreement* is that the paths agree at either side of this region. Our approach detects this.

To consider once again the family of paths $\pi(s_1), \ldots, \pi(s_k)$ through the same symbolic control-flow graph, we can apply our intersection algorithm to yield $T(\pi(s_1)) \cap \cdots \cap T(\pi(s_k))$. This intersection may have a set of path parameters $\langle i_1, \vec{\sigma_1} \rangle, \ldots, \langle i_p, \vec{\sigma_p} \rangle$, each of which representing a region of disagreement among the family of paths. By taking one such path parameter, $\langle i_1, \vec{\sigma_1} \rangle$, we can define

$$\{\pi : \pi \text{ is the subpath given to the path parameter } \langle i_1, \vec{\sigma_1} \rangle \text{ by some } \pi(s_j)\}$$

to be the set of subpaths taken in this specific region of disagreement by the paths in the family $\pi(s_1), \ldots, \pi(s_k)$. We can then invert the information that we have obtained to determine, for a specific subpath taken through a region of disagreement, which paths through the symbolic control-flow graph took that subpath. Ultimately, this leads to the ability to pair information from monitoring CFTL specifications with the subpaths through regions of disagreement that most often caused a certain monitoring result.

5.7 Paths and CFTL Specifications

We now consider how path reconstruction and comparison using the methods we have developed can be combined with the semantics of CFTL to give indications of why a program run might have failed to satisfy a CFTL specification.

Let us enumerate what we have so far. For a dynamic run $\mathcal{D}$ over a symbolic control-flow graph $\mathsf{SCFG}(P)$, we have a criterion called *branch-awareness* that ensures that we can

reconstruct the path taken by $\mathcal{D}$ through $\mathsf{SCFG}(P)$. Further, we have a notion of the path taken to reach a concrete state, which is still a path through a symbolic control-flow graph, but taking into account information from a dynamic run such as loop iterations. Finally, we can subtract one path from another and perform path intersections, which allow us to determine regions of code (via sections of paths in the symbolic control-flow graphs) that are present throughout a family of paths.

We will combine this approach with the CFTL semantics by first considering the falsifying concrete state that must exist in a violating dynamic run. Once we have this concrete state, it is always possible to determine the atoms in the CFTL specification that required information from it by using the $\mathcal{H}_\varphi^{-1}$ map that we developed in Chapter 4. For example, for a falsifying concrete state $\langle t, \sigma, m\rangle$, we can determine $\mathcal{H}_\varphi^{-1}(\sigma)$ which will give a set of pairs $\langle i_\theta, i_\alpha\rangle$ telling us the index in the set of static bindings and the index of the atom that required information from this concrete state. After doing this, we have that

$$\{\alpha : \langle i_\theta, i_\alpha\rangle \in \mathcal{H}_\varphi^{-1}(\sigma) \text{ and } \alpha \text{ is at index } i_\alpha \text{ in } A_\varphi\} \tag{5.3}$$

is the set of atoms to which the falsifying concrete state is relevant. The case of mixed atoms can be taken into account with a simple extension of the definition of this set to consider the elements of the domain of $\mathcal{H}_\varphi^{-1}$ which are maps.

5.7.1 Paths with Normal Atoms

To start with, we are interested in 1) isolating a single concrete state used to check the satisfaction of a CFTL specification and then 2) measuring the difference between paths taken to reach it. Ultimately, we can divide these paths up into those for which the CFTL specification was satisfied, and those for which it was not. Any characteristics we see in paths in either group will tell us something about that group.

The most straightforward examples of CFTL specifications are those with a single quantifier and single normal atom, so we will consider:

$$\varphi_1 \equiv \forall c \in \mathsf{calls}(\texttt{func}) : \mathsf{duration}(c) \in (0, 1) \tag{5.4a}$$

$$\varphi_2 \equiv \forall q \in \mathsf{changes}(\texttt{var}) : q(\texttt{var}) \in (0, 5) \tag{5.4b}$$

In the cases of these CFTL specifications, the set in (5.3) will only ever contain a single atom. We concentrate on (5.4b) for our first application of path comparison to CFTL specifications.

In this case, a falsifying concrete state must be a concrete state that satisfies $\mathsf{changes}(\texttt{var})$ because no temporal operators are applied to derive any other concrete states or transitions. Hence, we must think about the meaning of a path leading from the start of the symbolic control-flow graph to such a concrete state. Since we are placing a constraint over the values held in memory, the path that we reconstruct here gives an idea of the operations that take place over other variables that may lead to the computation of this value of `var`. It follows that a comparison of such paths may highlight differences in the computation taking place,

allowing us to isolate problematic branches with respect to the computation of the value held by `var`. The interpretation of what the difference in paths means depends on the specific use case, but here the ability to measure the difference in paths gives a clear way to identify problematic computations taking place. Conversely, we also have a way to isolate branches that do not contribute to the verdict. For example, if the path taken at a conditional differs but the verdict is the same, then we can conclude that this branching is unlikely to affect the result of computation.

If we consider φ_1, then the comparison of paths is likely to again give insight into computations of values, but this time when those values are used as parameters for calls of `func`.

We will now consider how path comparison can be used for more complex specifications for which more information is required from a dynamic run. Given the syntax of CFTL, this can happen in multiple ways.

5.7.2 Paths with Mixed Atoms

CFTL provides mixed atoms to allow the comparison of multiple quantities that can be measured at runtime. For example, sometimes comparison of the duration of a function call with a constant is not expressive enough; it is common for timing to depend on some property of some data.

With mixed atoms, and multiple measurements taking place at runtime (ie, values being taken from a dynamic run), this gives rise to a need for more complex path comparison in our explanation approach. To perform path comparison over paths derived for mixed atoms, we will use both path subtraction, and subsequent comparison (ultimately, we will compare the paths between two concrete states).

All specifications with mixed atoms use the same reasoning; the only thing that changes is which measurements are taken and what is done with them. Hence, without loss of generality, we consider the CFTL specification

$$\varphi \equiv \forall q \in \mathsf{changes}(\texttt{obj}) : \mathsf{timeBetween}(q, \mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))) \in (0, 2) \tag{5.5}$$

which expresses the property that the time taken for a commit to take place once `obj` has been modified should be less than 2 seconds. Here, the path taken to reach the call to `commit` from the change to `obj` is of particular interest, so being able to compare multiple paths across multiple dynamic runs would have clear utility.

Consider a dynamic run $\mathcal{D}$ that we check for satisfaction of φ. If $\mathcal{D} \not\models \varphi$ (and the partial semantics gives false), then there is necessarily a single falsifying concrete state. Further, the single atom can be split into two expressions, q and $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$, where $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$ must be in the future with respect to q by the definition of the next temporal operator. This means that a falsifying concrete state must be one that is identified by eval given the expression $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$. It follows that, if we let s be a concrete state for q and s' be a concrete state for $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$, then we are interested in the path $\pi(s') - \pi(s)$.

We now address the problem of comparing sets of instances of the path $\pi(s') - \pi(s)$ derived

from multiple dynamic runs, however we must first place an important requirement on the concrete states s and s'. That is, path comparison would not make sense if we were to compare paths between concrete states that corresponded to different symbolic states, so we require that the paths we compare start and end at the same symbolic states in the symbolic control-flow graph (even if they vary greatly between those two points).

To enforce this requirement, we must carefully select the dynamic runs that we include in our analysis. Given a family $\mathcal{D}_1, \ldots, \mathcal{D}_n$ of dynamic runs and the CFTL specification φ, we select a $\mathcal{D}_i$ such that $\mathcal{D}_i \not\models \varphi$. If $\mathcal{D}_i \not\models \varphi$ (again, with the partial semantics also giving false; this will be assumed), then there is a falsifying concrete state in $\mathcal{D}_i$ and, given φ, it must be for the expression $\mathsf{next}(q, \mathsf{dest}(\mathsf{calls}(\texttt{commit})))$.

Once we fix a symbolic state based on the falsifying concrete state to which paths will go, we must fix the symbolic state from which paths will leave. Since the same symbolic state can be identified based on the same atom for multiple static bindings (ie, $\mathcal{H}_\varphi^{-1}(\sigma)$ gives a list of pairs, multiple of which have the same atom index but with different static binding indices), we must fix a static binding. This will implicitly fix a symbolic state for the expression q. To see this, consider the code:

```
obj = obj1()
obj = obj2()
commit()
```

Here, two static bindings will be constructed by the quantifier predicate $\mathsf{changes}(\texttt{obj})$ but both will identify the state after the instruction `commit()` as being required for the expression $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$. Hence, after fixing the symbolic state for the falsifying concrete state, we must also decide whether to take the symbolic state generated by `obj = obj1()` or `obj = obj2()`. The choice depends on the context in which we perform the analysis, but it must be consistent throughout in order to fulfil the requirement that paths in our comparison must start and end at the same place. Once the requirement is fulfilled, we can go on with our analysis.

Suppose that we fix symbolic states σ and σ' for q and $\mathsf{dest}(\mathsf{next}(q, \mathsf{calls}(\texttt{commit})))$ respectively. Then, taking the family $\mathcal{D}_1, \ldots, \mathcal{D}_n$ of dynamic runs, we first find a $\mathcal{D}_i$ with $\mathcal{D}_i \not\models \varphi$ and such that there are concrete states whose corresponding symbolic states are σ and σ'. If we cannot find such a dynamic run, then we choose different symbolic states. Finally, with our fixed symbolic states σ and σ', we construct the family of dynamic runs from the existing one, but ensuring that dynamic runs kept are those with concrete states corresponding to the symbolic states σ and σ'. Our new family of dynamic runs is therefore $\mathcal{D}_{k_1}, \ldots, \mathcal{D}_{k_n}$ where the k_i are indices from the original family. With this new family, we can compute the set

$$\{\pi(s') - \pi(s) : s, s' \text{ are in some } \mathcal{D}_{k_i} \text{ where } s \text{ has symbolic state } \sigma \text{ and } s' \text{ has } \sigma'\}.$$

Since these paths start and end at the same point (s and s' must all have the symbolic states σ and σ' respectively), we can compare them using our path intersection approach. To do this, we map the set we have to the set of parse trees, and then run the intersection algorithm on this set.

If the result of this intersection is a parametric path, then we care about what the values of

the path parameters were when φ was satisfied and violated. The outcome of this check would show that either 1) the branch taken between the two concrete states required for the CFTL specification does not matter; or 2) a certain branch is more likely to cause a problem.

5.7.3 Paths in the Interprocedural Setting

If one were to place a constraint over a call to some function f using CFTL, it would be natural to link the results of this measurement to the path taken inside f. We have already discussed the reconstruction of entire program paths, so we already have the machinery to instrument both the function that calls f and f itself.

Now, dynamic runs encode function calls by transitions. This means that, if another dynamic run were to represent the computation taking place further down the call chain, the timestamps associated with its concrete states would have to *fit inside* the transition. So far, our representation of dynamic runs has been with time that started from zero; we have no notion of a *shared clock* among multiple dynamic runs that would allow us to say that one dynamic run occurs during another (and even during a transition occurring during another). This means we must extend our notion of a dynamic run to the interprocedural setting in which multiple dynamic runs are present and the time associated with them is taken from a single, central clock.

We remark here that, because dynamic runs require instrumentation, any functions that are not instrumented do not generate dynamic runs. Because of this, we cannot say that a dynamic run that fits inside a transition of another is called *directly*, only that it is below the caller in the call chain.

We now introduce the notion of an *interprocedural dynamic run*, which combines a set of dynamic runs with their own local time (encoded in the timestamps of their concrete states) and a global clock, which maps concrete states from each dynamic run to a global time.

Definition 7. *An* interprocedural dynamic run *is a pair* $\langle \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}, \Phi \rangle$ *where the* $\mathcal{D}_i$ *are dynamic runs and* Φ *is the function*

$$\left(\bigcup_{\mathcal{D}_i} \mathsf{states}(\mathcal{D}_i) \right) \to \mathbb{R}^{\geq}.$$

We call Φ the *global clock* and use it to map concrete states (whose timestamps are in *local time* with respect to their dynamic runs) to a global time. This mapping allows us to say that one dynamic run occurs during another; the exact mechanism by which concrete states are associated with a global clock is left to implementation. Finally, we observe that instrumenting a transition $\langle s, s' \rangle$ requires including the symbolic states of s and s' in the set of instrumentation points meaning that, even after instrumentation, the concrete states at both sides of a transition are present and will be given global timestamps by the global clock.

With a global clock attached to a set of dynamic runs, it is straightforward to determine whether a dynamic run occurs during a transition $\langle s_1, s_2 \rangle$. Further, it is straightforward to determine whether a dynamic run represents the call that is next in the call chain after a caller. Assuming that the *minimal concrete state* of a dynamic run is that which is minimal with

```
def caller():
  [...]
  f1()
  [...]

def f1():
  f2()
  f3()

def f2():
  [...]

def f3():
  [...]
```

Figure 5.13: A simple system of multiple functions.

respect to the time-induced total ordering on the concrete states of that dynamic run, and similarly for the *maximal concrete state*, the steps are as follows:

1. We find a set $\{\mathcal{D}_i\}$ such that each $\mathcal{D}_i$ whose minimal $s_3 \in \mathsf{states}(\mathcal{D})$ and maximal $s_4 \in \mathsf{states}(\mathcal{D})$ have $\Phi(s_1) < \Phi(s_3) < \Phi(s_4) < \Phi(s_2)$.
2. We find the unique $\mathcal{D}^* \in \{\mathcal{D}_i\}$ for which the minimal $s_3 \in \mathsf{states}(\mathcal{D}^*)$ has $\Phi(s_3) < \Phi(s_3')$ for all s_3' minimal in the other dynamic runs.

To see why we consider a set $\{\mathcal{D}_i\}$, consider the example of a simple system written in Python with multiple functions in Figure 5.13. Suppose that the functions `caller`, `f2` and `f3` are the only ones to be instrumented, therefore also the only ones to generate dynamic runs. Then the dynamic runs generated by the calls to `f2` and `f3` would both fit inside the transition generated by the call to `f1` during a call to `caller`.

In the case of recursion (for example, a monitored function `f` calling itself multiple times), then we would only care about using the information obtained during each call to explain behaviour in the call above.

Comparing Paths

So far, we have considered the comparison of paths leading to an observation inside a single scope. With the development of the notion of an interprocedural dynamic run, we can easily compare paths taken inside function calls over which we place constraints.

We highlight that we only consider the interprecodural setting in the context of explanation, and not in the context of specification, because this would require an extension of the CFTL syntax and semantics. While such an extension is of course feasible, we have not needed to do it for the application described in this thesis.

Procedures across Machines

We now briefly introduce the theoretical foundation of a topic covered in Chapter 6: analysing information taken from monitoring function runs on different machines. The problem is essentially the same as for multiple procedures, except the requirement for a global clock is more important. For example, in the interprocedural setting, the *global clock* is obtained simply by relying on the time held by the machine on which the monitored code runs. However, when functions are running on separate machines we can no longer rely on system time being synchronised, so the requirement for a global clock is more important in implementation.

The theoretical setup is isomorphic to that introduced in Section 5.7.3; monitoring across multiple machines will lead to a set of dynamic runs $\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$, and the synchronisation mechanism Φ is some central clock server.

5.8 Concluding Remarks

We have now introduced our theoretical contributions to the problem of determining some indication of why a dynamic run failed to satisfy a CFTL specification. In doing this, we have introduced a general approach for the comparison of program paths consisting of 1) translating the program's symbolic control-flow graph to a context-free grammar; and 2) applying an intersection algorithm to parse trees derived with respect to this grammar. Our intersection approach allows differences in paths, such as branches taken through conditionals, to be determined.

We have used our general program path comparison approach to compare the paths taken through symbolic control-flow graphs by dynamic runs. Further, we have provided a way to determine the program path leading up to a given concrete state (normally, a *falsifying concrete state*) in a dynamic run that contains sufficient information. Given this link between dynamic runs and paths through symbolic control-flow graphs, we have shown how differences in program paths can be used to indicate *explanations* of false verdicts.

Finally, in Chapter 6, we will introduce our analysis tools which exploit the theoretical foundations that we have introduced.

Chapter 6

Implementation

All of the theory described so far in this thesis has been implemented in the publicly available VyPR[1] framework, whose project website can be found at `http://pyvypr.github.io/home/`. Instructions on downloading VyPR and other tools in the ecosystem, along with instructions on writing specifications, using the web-based analysis tool and writing analysis scripts, can be found at `http://pyvypr.github.io/home/use-vypr.html`. Further, the entire VyPR ecosystem can be downloaded in the form of a Docker container at `http://pyvypr.github.io/home/rv2020-tutorial/`.

The implementation available online is an extended version of what was described across two papers presented at TACAS 2019 [DRF+19] and CHEP 2019 [DHR+19], along with an extended tutorial at RV 2020 [DHJ+20]. VyPR was developed in order to fulfill the use cases present in software both within CERN and further afield. It is designed to provide sophisticated analysis techniques for software engineers working on normal Python programs, web services and systems with a server and client, while requiring a minimal learning curve. This chapter describes the implementation of the group of tools that make up the VyPR project.

6.1 A Hierarchy of Programs

We consider two main categories of programs:

Long-running programs whose computations can be grouped by requests for data coming from other processes.

Short-running programs whose computations are all performed with respect to a single set of inputs given at the start of program execution.

In this thesis, we focus on long-running programs and, in particular, *web services*. In such cases, the requests for data are usually HTTP requests, which consist of 1) a message being received via some network communication mechanism such as a socket; 2) subsequent computation which typically varies with the request contents; and 3) a response.

[1] VyPR stands for **V**erif**y** **P**rograms at **R**untime.

Web services are of course not the only programs to be considered as long-running; one might write a program with the intention of daemonising it to have a background process that is constantly working and ready to respond to some external input. However, we restrict ourselves to the web service setting because of our use cases.

The base form of VyPR is a tool that can be used for analysis of Python programs in general. However, to give engineers a more intuitive interface, there is one release for web services and another for short-running programs. We also remark that this thesis focuses on single-threaded programs and, even in the multithreaded case, analysis is only performed per-thread, rather than considering phenomena resulting from multithreaded programs. However, given that case studies that are planned as future work use multiple threads, expansion to deal with this setting would be a sensible next step after the work in this thesis.

Finally, all of the material that is presented in this chapter could be generalised to longer running programs that are not web services, but the case studies performed were on web services so we stick to these.

6.2 The VyPR Project

The VyPR project consists of multiple tools written in Python that aim to make performing sophisticated analysis of Python programs more straightforward for engineers. We will now give detail to each of these tools, before discussing the problem of designing them in a way that is accessible for engineers who have no desire to learn the formal mathematical foundations presented so far in this thesis.

6.2.1 Core

The central tool that supports all of the other tools in the ecosystem is the VyPR core. The release for web services is found at `http://github.com/pyvypr/VyPR/` and the release for short-running programs at `http://github.com/pyvypr/VyPRLocal/` (Python 2 and 3 variants are found within the same GitHub organisation). Both include 1) the library with which engineers can construct specifications; 2) the instrumentation machinery (based on the material in Chapter 4); and 3) monitoring machinery (also based on the material in Chapter 4). All implementations of VyPR so far require engineers to write specifications over individual functions in their programs; interprocedural and inter-machine analyses are possible in post-processing of verdict information from monitoring of functions in isolation. All other tools in the ecosystem are completely general and do not have releases for different categories of programs. This is because they only deal with the data generated by VyPR, whose schema is sufficiently generalised across multiple categories of programs, whereas the VyPR core has to deal with the vastly different architectures.

6.2.2 Verdict Server

The information obtained during instrumentation and monitoring by the VyPR core must be stored persistently somewhere. The first, immediate reason for this is that the information must

be persistent for an engineer to be able to perform analysis. The second is that instrumentation must be done in a separate process (especially for shorter running programs, instrumentation for every run would be unnecessary overhead), so the information must be made persistent and accessible during monitoring. Found at `http://github.com/pyvypr/VyPRServer/`, the VyPR server is written in Python using Flask [pytb] and is communicated with by HTTP.

Aside from making data from instrumentation and monitoring persistent, the verdict server caters for multiple use cases:

- It provides an API for the Python-based analysis library (Section 6.6). This API wraps common queries that would otherwise be complex in simple calls.
- It provides a web-based analysis environment (Section 6.8) that goes even further than the analysis library by attaching many complex queries to interactions with a Graphical User Interface. This environment is also served by a separate API written specifically to return data in a format that is useful in the web application setting.

6.2.3 Analysis Library for Python

Part of the VyPR ecosystem is the Python-based analysis library whose source code is found at `http://github.com/pyvypr/VyPRAnalysis/`. The library is intended for use in analysis scripts (it is used extensively in Chapter 7) to remove the need to know precise details of the structure of the performance data held by the server. In particular, the relational schema used to store the data is designed to reduce space consumption by minimising duplication of data. Without this library deployed alongside VyPR, the information stored on the verdict server would be difficult to interpret.

Further, the analysis tools are used for prewritten scripts distributed with the verdict server. These scripts (see Section 6.7) add to the theory presented on explanation in Chapter 5, by performing path-based analyses over data stored on the verdict server for certain categories of CFTL specifications. Without the analysis tools built around VyPR, these prebuilt scripts would be much more complex than they are, especially the parts involving program path comparison and call tree reconstruction.

6.3 Runtime Verification in Practice

Before describing the architecture of VyPR in depth, we make some comments on the problem of marketing a tool that allows analysis of programs by application of a formal method. While Runtime Verification is not a *heavy-weight* formal verification method (and should be seen to go alongside, and certainly not replace, testing), its formal foundations can be a deterrent to engineers wishing to analyse their programs with no significant (and certainly not formal mathematical) learning curve.

The first insight into this problem is that it appears to be best to avoid referring to VyPR as a *Runtime Verification tool*. Any reference to formal verification, or formal methods, to software engineers can carry negative implications of a requirement to understand heavy mathematical foundations. VyPR is therefore advertised as a tool with which one can *query* the performance

of their programs and inspect the results. Despite the specification language CFTL being a temporal logic with rigorous mathematical foundations, none of this is mentioned and we instead refer to writing *queries* over a program in an *intuitive query language*. As a note to the reader, despite our modified language when communicating outside the formal setting, this is the only time we will use this language and, for the rest of the thesis, the usual (such as *specification* rather than *query*) will be used.

The formal internals of instrumentation are also hidden, and we refer to the fact that VyPR can automatically determine from where in a program to take data given a query written by an engineer. Once monitoring has been performed at runtime, the analysis library that we distribute for VyPR, found at `http://github.com/pyvypr/VyPRAnalysis/`, makes inspection of the data stored by VyPR at runtime much simpler than it would be if one would have to query the data directly. Further, the web-based analysis tool that is deployed with the verdict server abstracts even further away from the VyPR internals, allowing detailed investigations of data without the need to write any code.

Finally, our discussions with engineers at CERN led us to shape the development of VyPR in order to increase its utility to our case study. The most valuable input was given by 4 software engineers with Physics and Computer Science backgrounds across 3 teams, each engineer having at least a Masters degree. In particular, there was one engineer working on the main case study; one engineer working on another project whose feedback led us to realise that a future extension of CFTL to the inter-procedural setting would be useful; and two engineers on a final project whose feedback led to the development of *specification compilation* (see Section 6.4.2).

6.3.1 The Philosophy of Minimal Intrusion

Practical experience has shown us that an analysis approach that requires a significant amount of modification to the monitored source code will not be adopted by engineers for projects that consist of more than a single Python module. The design goal of VyPR was to make the tools supported by the theory in this thesis accessible to engineers with minimal intrusion into the source code of their projects.

With this in mind, VyPR allows engineers to monitor their projects with (usually) only a couple of lines of code to *activate* VyPR's monitoring machinery and a specification written separately from their projects. Instrumentation and monitoring are performed with no input from the engineer, and the first steps have been taken to generate *explanations* without any intervention.

6.3.2 A Specification Language for Engineers

Control-Flow Temporal Logic (CFTL) was already designed to be an intuitive specification language, but there is also responsibility placed on the way in which a CFTL specification is given to VyPR since this is another language in itself (based on the mathematical formalism).

To reduce the learning curve for engineers using VyPR, we opted to allow input of CFTL specifications by means of a Python library. The result of this is that a specification would ultimately conform to Python syntax while using calls to a library to construct the specification

structure in memory for VyPR to use. This decision also means that we can rely on the stability of the Python parser, instead of writing a new parser for a new language.

Inline with the decision to hide the formal mathematical foundations from the engineer, VyPR's specification writing library is designed with the intention of preventing engineers from feeling like they are writing mathematics. Engineers should see the specification process as the task of writing a query over their program; this is helped by the low level of CFTL.

6.4 The Architecture of VyPR

We now describe the separate components of the VyPR core. Variations of each component are used to cater for the web service and short-running program cases.

6.4.1 Building Specifications

An engineer must be able to easily express the properties that their program must hold, but instrumentation and monitoring given such properties must also be possible. The solution provided by VyPR is the PyCFTL library, which meets two requirements:

- The specifications written with it are tightly coupled with the source code being analysed. CFTL, unlike higher-level specification languages, does not need an instrumentation mapping to relate the program source code to the symbols used in the specification. Further, since symbols in CFTL specifications are derived from the program source code itself, only the source code is needed for a specification to make sense. These facts are preserved by the Python library provided by VyPR.
- A single structure is generated in memory that VyPR can easily use to 1) perform instrumentation and then 2) monitoring of a program.

The approach taken in the design of PyCFTL is to make the code written as similar to reading something in natural language as possible. We start with an advantage because Python's syntax is already very friendly, but we aim for a library in which writing *queries* over quantities that can be measured at runtime is straightforward.

We begin a description of PyCFTL by expressing a performance property in natural language, then in CFTL and finally in PyCFTL code. Suppose that we want to express the very simple property that, during an execution of a specific function, all calls to the function `execute` should take no more than 1 second. Then one could write the CFTL specification

$$\forall c \in \mathsf{calls}(\texttt{execute}) : \mathsf{duration}(c) \in [0, 1].$$

One would construct the PyCFTL code to express the same property in two steps. First, the quantifier would be constructed by creating an instance of the `Forall` class. Specifically, one would write `Forall(c = calls("execute"))`. Then, the quantifier-free part would be written using two further steps. The first would involve constructing a *lambda*[2] expression that takes

[2] An anonymous function whose implementation in Python can be used, as is the case in VyPR, to lazily evaluate expressions.

the variable `c` (this must match the variable used in `Forall`). The result would be the code:

```
lambda c : c.duration()._in([0, 1])
```

Here, `c` is assumed to be an object representing a function call. This representation is made clear by the definition of the method `duration` on `c`. Finally, the lambda expression would be associated with the quantifier we built by writing:

```
Forall(c = calls("execute")).\
Check(lambda c : c.duration()._in([0, 1]))
```

We make a key observation on this code: the names defined in the quantification, in this case only `c`, are treated as the *events* that they represent at runtime inside the lambda expression. For example, since `c` is an object representing a call of the function `execute`, we can measure the duration of the function `execute` by calling the method `duration()` defined on `c` and then continue by placing a constraint over that quantity.

This idea of variables holding objects that represent quantities measurable at runtime is critical in the design of PyCFTL. We will now investigate some more examples which will show how this system can be used to easily write complex constraints over multiple quantities measurable at runtime.

Quantifying over Concrete States

Suppose we would like to express the CFTL specification

$$\forall q \in \mathsf{changes}(\texttt{result}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{verify}))) \in [0, 0.5].$$

Then we take the same approach as in the previous example and use `Forall` to set up a quantifier: `Forall(q = changes("result"))`. We then have a variable `q` that will hold a *concrete state* from which we would like to find the next call to `verify` and measure its duration. We can use the `next_call()` method defined on `q` to give an object representing the next call to `verify` after `q` (the analogue in a dynamic run is the next transition). On the resulting object, we can again call the `duration()` method. Together, this gives:

```
Forall(q = changes("result")).\
Check(lambda q : q.next_call("verify").duration()._in([0, 0.5]))
```

Multiple Quantifiers

Suppose we would like to express the following CFTL specification with multiple quantifiers and a more complex inner structure

$$\begin{aligned}&\forall q \in \mathsf{changes}(\texttt{auth}) :\\ &\forall t \in \mathsf{future}(q, \mathsf{calls}(\texttt{execute})) :\\ &\qquad q(\texttt{auth}) = \texttt{True} \implies \mathsf{duration}(t) \in (0, 0.1).\end{aligned}$$

A natural language interpretation of this is that we require, for each change to `auth`, that every future call of `execute` should take strictly less than 0.1 seconds if `auth` is `True`. Expressing this with PyCFTL is straightforward; one simply chains together multiple calls to `Forall`:

```
Forall(q = changes("auth")).\
Forall(t = calls("execute", after="q")).\
Check(lambda q, t :
  If(q("auth").equals(True)).then(
    t.duration()._in((0, 0.1))
  )
)
```

In this case, we make use of the `If(...).then(...)` syntactic sugar which generates a disjunction based on the identity $p \implies q \equiv \neg p \vee q$. The alternative form of the inner-part of the specification without syntactic sugar would be:

```
lor(
  lnot(q("auth").equals(True)),
  t.duration()._in((0, 0.1))
)
```

We also make use of the `equals()` method defined on state values (to use this, a value must first be accessed by treating `q` as a function with the syntax `q("auth")`).

Comparing Multiple Quantities

As shown in Chapter 3, CFTL supports comparison of two measurable quantities with each other. This greatly increases the expressive power of the specification language, and is implemented in VyPR's specification library so as to be easily exploited.

Suppose that we would first like to express the requirement that the time taken by a function `check` is less than the length of the list `items` (only known at runtime) multiplied by a factor of 0.01. Then the CFTL specification would be as follows:

$$\forall c \in \mathsf{calls}(\texttt{check}) : \mathsf{duration}(c) < \mathsf{length}(\mathsf{source}(c)(\texttt{items})) \times 0.01$$

A more mathematically-oriented description of this specification could be "for every call of `check`, the duration of that call should be strictly less than the length of the variable `items` in the state immediately preceding the call, multiplied by 0.01". This could be written in VyPR's specification library as such:

```
Forall(c = calls("check")).\
Check(lambda c : c.duration() < c.input()("items")*0.01)
```

As usual, we treat `c` as a variable holding a function call and measure its duration using the `duration()` method. We then construct a comparison by using the overloaded `<` operator. On the right hand side, we refer to the state immediately before the call to `c` using the `input()` method, and then treat that expression as a state whose value of `items` we can obtain by writing `c.input()("items")`. VyPR's specification library then supports arithmetic on measured quantities (as long as the constant is on the right of the arithmetic operator). One can also perform multiple stages of arithmetic, such as `(c.input()("items")*0.01)+1` and VyPR will maintain a stack of arithmetic functions that are applied to the relevant quantity when it is measured.

Measuring Time Between States

It is a reasonable assumption that an engineer may want to measure the time taken to reach one concrete state from another. For example, there may be some branching performed between the end of a call to the function `construct` and the end of the next call to the function `commit`. An engineer may be curious if one path taken can take much longer than another, with respect to a baseline of 0.5 seconds. To measure this, one could write the CFTL specification:

$$\forall c \in \mathsf{calls}(\texttt{construct}) : \mathsf{timeBetween}(\mathsf{dest}(c), \mathsf{dest}(\mathsf{next}(c, \mathsf{calls}(\texttt{commit})))) < 0.5.$$

VyPR's specification library provides the timeBetween operator in an identical way to CFTL, so the specification could be:

```
Forall(c = calls("construct")).\
Check(lambda c :
  timeBetween(c.result(), c.next_call("commit").result()) < 0.5
)
```

In this case, we use the `result()` method to obtain the state attained immediately after a function call. The decision to translate the mathematical dest and source operators to the `result()` and `input()` methods in the specification library was taken to make specifications given to VyPR closer to source code, rather than the mathematical abstraction of dynamic runs.

6.4.2 Compiling Specifications

Given a specification that an engineer has written, there must be an intuitive way to indicate to VyPR which functions in a system should be monitored with respect to that specification (from a theoretical point of view, which dynamic runs should be checked). The solution implemented in VyPR is *compilation* of specifications. The idea is for engineers to use a specific format to specify the set of functions in which monitoring (and also instrumentation) should occur. For example:

```
{
  "app.views" : {
    "store_blob" : [
      Forall(c = calls("store")).\
      Check(
        lambda c : c.duration() < c.input()("blob").length()*0.001
      )
    ]
  }
}
```

Here, `app` is the top-level package, `views` is a module inside that package and `store_blob` is a function inside that module. Hence one could refer to the function `store_blob` using the fully-qualified name `app.views.store_blob`.

However, suppose the engineer wishes to apply a specification in every place in their program at which a function with the name *check* is called. Taking only the specification format listed

above would require the engineer to inspect their code themselves and construct the appropriate dictionary mapping each relevant function to the same specification.

VyPR removes the need for this by adding a compilation layer to the way that specifications are constructed. That is, engineers can use library functions provided by PyCFTL to build selection criteria to determine the set of functions at which they would like to monitor for a property. One could write, for example:

```
{
  Functions(containing_call_of="check") : [
    Forall(c = calls("check")).\
    Check(
      lambda c : c.input()("data").type() == c.result()("data").type()
    )
  ]
}
```

which would cause VyPR to make sure that, wherever `check` was called, the type of the value given to `data` does not change. We remark that this specification only captures calls of *some function* which is locally bound to the name `check`, since our instrumentation uses only the information it finds immediately in code and we have not yet needed to extend it.

Ultimately, the additional compilation layer for specification addresses a problem found a lot when addressing real use cases: it can be difficult to know where to apply specifications. This is a problem that we were made aware of when discussing VyPR with other teams around CERN. Using the compilation layer, engineers can either write their entire specification based on these function selection criteria, or use them as a starting point to identify problematic behavior in regions of their system.

In relation to other languages, the compilation layer could be seen to be similar to the notion of an *Aspect* [Asp] used in Java instrumentation. We acknowledge that Aspects in Java can operate at runtime (as well as statically), whereas the facility provided by PyCFTL relies solely on static analysis. However we make two key observations:

- The static analysis we currently use is basic and extension would lead to more cases being captured.
- Our use cases so far have not shown a need for more sophisticated static analysis.

Finally, static analysis for instrumentation is also employed by the TraceMatches [BHL+07] tool, which divides instrumentation points into *probes* and runs multiple instances of the program under scrutiny, each with only one of these probes instrumented.

Optimising Compilation

This compilation is implemented by constructing the symbolic control-flow graph of the functions in a system and searching their vertices to determine whether any satisfy the criteria given in the specification. This is a potential source of inefficiency which could be removed by caching a map from functions in a system to the variables/functions on which they operate.

6.4.3 Instrumentation

Once a specification has been written using VyPR's library, the instrumentation process can inspect the structure it constructs in memory and insert instruments according to the strategy described in Chapter 4.

Recall that the first stage of our instrumentation strategy involves inspecting the sequence of quantifiers. During such inspection, a space of maps is constructed from variables in the specification to edges and symbolic states from the symbolic control-flow graph. In practice, we first compute a map $\mathsf{reachable}(\mathsf{SCFG}, \sigma)$ that gives the set of all symbolic states and edges reachable from the symbolic state σ in the given symbolic control-flow graph. This is straightforward; we perform depth-first search of the graph for each of its symbolic states. Then, we apply Algorithm 4 to the sequence of quantifiers, using reachable to amortise any reachability computations we need to perform. Finally, Algorithm 5 is used to determine the instrumentation points for each static binding. If the software engineer wants to use the explanation machinery in VyPR, the instrumentation process will also place path recording instruments (as described in Chapter 5).

Now, since only indices are used in the optimised monitoring algorithm (Algorithm 7), we have no need to pass any actual objects that we found to the monitoring algorithm. The only major problem to solve is that of placing the instrumentation code.

The solution employed by VyPR is to use Python's native Abstract Syntax Tree library `ast` to determine the abstract syntax tree of a target program. Once this has been computed, we use the same library to determine the abstract syntax trees of the relevant instrumentation code and insert it at the points dictated by our traversal of the symbolic control-flow graph (to do this, we have a map from edges and symbolic states to parts of the program's abstract syntax tree). Once the insertion is complete for all instrumentation points, we compile the final abstract syntax tree to Python bytecode and write to a bytecode file, renaming the original so recompilation does not happen during Python's runtime. This way, we prevent the Python interpreter's default behaviour which is intended to make sure that all bytecode processed at runtime reflects the source code, if it exists. With the only version of a module available being the instrumented bytecode version, Python imports that at runtime, allowing monitoring to happen.

6.4.4 Online Monitoring

Monitoring is performed by VyPR in the same process as the monitored program, but in a different thread. This ensures that the monitored program can continue if there is a problem with the monitor. Further, we assume that the program being monitored has a single thread. If there are multiple threads then, as mentioned before, we consider each in isolation. Instrumented code interacts with the monitoring thread by means of an intermediate consumption queue to which instruments placed by VyPR add information.

Structuring Data from Monitoring

Monitoring in VyPR generates verdicts in a slightly different way to that which is described by the semantics in Chapter 4. In particular, instead of generating a single verdict that is obtained by monitoring the target program with respect to a property, a verdict is reached for each binding generated by the sequence of quantifiers. In offline analysis, this layout of information allows more precise querying. For example, engineers can inspect the truth value and the observations needed for each binding rather than simply knowing whether the *program run as a whole* satisfied the specification.

Now, suppose an engineer wants to monitor a program for agreement with the specification

$$\forall c \in \mathsf{calls}(\texttt{construct}) : \mathsf{timeBetween}(c, \mathsf{dest}(\mathsf{next}(c, \mathsf{calls}(\texttt{commit})))) < 0.5$$

with the Python code for the specification being:

```
Forall(c = calls("construct")).\
Check(lambda c : timeBetween(c, c.next_call("commit").result()) < 0.5)
```

Further, suppose that the program under scrutiny has multiple calls of `construct`. Then a dynamic run that includes more than one of these calls will generate more than one binding based on the quantifier $\forall c \in \mathsf{calls}(\texttt{construct})$.

Moving our attention to a single binding derived from a dynamic run: for each atom in the inner-part of the specification, evaluation requires either one or two concrete states or transitions. In implementation, we construct a map from atom/sub atom index pairs to concrete states/transitions. Recalling that we use formula trees to keep track of the verdict so far for a given binding, we also record the atoms that caused the final collapse of a formula tree.

We now consider how dynamic runs are represented in implementation. Given that VyPR performs monitoring over individual function runs, these are a natural choice to be the practical analogue of dynamic runs. Further, we group dynamic runs, or function calls, by *transactions*. We use this term with the aim of being as general as possible. For web services, transactions are HTTP requests during which there can be multiple function calls and, for local programs, they are individual program runs. Finally, the analogue in implementation of a concrete state and a transition is an *observation*, with which we associate a start and end time, and whichever quantity was relevant for the measurement that took place at runtime.

Asynchrony in Python

We now make some comments on the architecture of VyPR with respect to overhead and the limitations imposed on us by the standard Python interpreter.

An important factor to consider when measuring the overhead induced by a tool that *attaches* to a Python program by setting up a separate thread and performing some computation on information received from it is the *Global Interpreter Lock* (GIL). This is the mechanism used by the default Python interpreter (CPython [cpy]) when dealing with multiple threads sharing memory.

In most experiments described in this thesis, we are using a version of Python 2 in which the Global Interpreter Lock means most operations, even if set up in Python code to be performed

asynchronously with threads, are essentially run in sequence. This is a consequence of the GIL's policy of giving a single thread exclusive control, which is a simple way to avoid race conditions that are typical in concurrent systems, but prevents true concurrency for anything other than IO. Most attempts to write asynchronous code with CPython's native threading library will in fact lead to serial code which is possibly less efficient than its intentionally serial counterpart because of the overhead of thread management.

This is an observation that we must take into account with VyPR, since currently no work has been done to overcome this limitation imposed by the GIL. In fact, in our SAC-SVT paper [DR19b], we highlight that using VyPR to monitor programs with little to no IO can lead to dramatic increases in overhead. However, we finish this remark with the further observation that the applications we consider give some time for VyPR's monitoring thread to actually check the data it receives from the monitored program. In web services, this time often comes during the IO required for HTTP requests and remote database operations, which both involve writes to sockets.

Alternatives to Asynchronous Monitoring

Python is a unique language in how many tools it offers to engineers to build tools such as debuggers and profilers. In particular, the standard Python interpreter offers *tracing* functions (sometimes colloquially referred to as *trampolines* because of a particularity in how engineers must implement them) which are called by default at runtime for most events (such as function calls, line executions and function returns) occurring at runtime.

Tracing functions are a natural alternative to VyPR's approach to monitoring by starting a separate thread, but there is one important reason for which we elected not to use them: one must still make a decision for many events occurring at runtime about whether that event is relevant to the property being monitored. While tracing functions offer a method of *dynamic instrumentation* (modifying instrumentation at runtime according to value changes, such as primitive to reference or vice versa), the underlying machinery means they can induce much more overhead than we want. Further, their naturally synchronous nature means any slow-down they induce directly affects the monitored program.

Implementation of Weak vs Normal Temporal Operators

We finish with the remark on how the VyPR monitoring algorithm reflects the theoretical details given so far. The formal semantics of CFTL distinguishes between *weak* and *normal* versions of operators such as `next` by changing the verdict if nothing is observed for them. However, VyPR currently disregards weak versus normal temporal operators in favour of the policy that, if nothing is observed, no verdict is generated (rather than generating true or false verdicts retrospectively).

6.5 A Repository for Verdict Data

The central server provided alongside VyPR (see `http://github.com/pyvypr/VyPRServer`) acts as a central repository for possibly multiple instances of Python programs being run with

VyPR attached. This server is responsible for storing verdict information, as described in Section 6.4.4, in a format that is space-efficient. There is further pressure on the central server to offer a storage format on which IO operations are fast. Currently, this format is a relational schema with tables to cater for every part of the monitoring data, including dynamic runs, bindings (static bindings, and those derived from a dynamic run), atoms (and their observations and relevant instrumentation points) and path information for our explanation approach.

All of this information is accessed via an analysis API provided by the front-end of the server. The API wraps up the common tasks one could perform with verdict data into simple calls and hides the underlying database. This is used by the analysis tools, leading to large simplifications in the analysis process seen by engineers. An API is also provided for the VyPR core to store data obtained during instrumentation and monitoring.

6.5.1 Storing Path Information

Recall that a *branch-aware dynamic run* is intuitively a dynamic run in which there is no ambiguity of the path to take through the symbolic control-flow graph to reach one concrete state from the one immediately preceding it in time. While VyPR's instrumentation strategy will ensure that this is the case, the only concrete states that are stored in the database are those that are needed to determine truth values of the atoms in CFTL specifications; a different approach is taken to store path information.

We introduce the notion of a *path condition chain*, which represents the path taken by a dynamic run as a sequence of path conditions associated with a dynamic run in the verdict schema. To save space, the actual path conditions are stored in a separate table and the sequence stored with a dynamic run contains IDs.

Further, multiple concrete states that were needed to determine the truth values of atoms can be linked to the path taken by the dynamic run to reach them without duplicating path information. Given a concrete state, this is done by associating with it the offset into the relevant path condition chain. However, this only allows approximate reconstruction of the path; if there are multiple concrete states generated by a region of code with no branching, they will all link to the same path condition in the chain. This is shown by the code below, at lines 7 and 8. Each of these lines can be reached by satisfying path condition $\neg$ `P` and then `Q`:

```
1  if P:
2    a = 10
3    b = 20
4  else:
5    c = 10
6    if Q:
7      d = 20
8      f()
9    else:
10     e = 30
11     f()
```

Our solution is to associate with every concrete state a row in the *instrumentation point* table in the relational schema. The row associated with a given concrete state contains a *reaching path length*. This quantity represents the length of the path from 1) the symbolic state at

which the most recent path condition was satisfied up to 2) the symbolic state associated with the concrete state. The result of such a quantity is the removal of ambiguity when multiple concrete states link to the same path condition. Using this approach, even though lines 7 and 8 in the example above can be reached by satisfying the same sequence of path conditions, they can be distinguished by their distance from the most recent (ie, closest backwards reachable) statement in the program at which a decision was made. Hence, we associate line 7 with the sequence of path conditions $\neg$ `P`, `Q`, but also the *reaching path length* 1, and then the same for line 8 but with reaching path length 2. The addition of this reaching path length value removes the ambiguity.

We also note that, for any function and specification combination, the way in which we store the path taken is independent of the size of the specification. This is because only one path is stored for each call of the function and the cost of linking concrete states to the correct path condition in the sequence is small.

A Remark on Efficiency

Storing paths as *flattened* sequences of path condition IDs allows insertion of paths to be performed within just one database query. The first approach to storing path information was to use a different row for every element in the path condition sequence. This allowed space efficient mapping of observations to the point in a program path at which they occurred, but was inefficient when path insertion was performed. In particular, when storing path information the worst cases tend to involve loops. If a loop completes hundreds of iterations or more, then a single insertion query would be required for every iteration of the loop. This does not scale to systems whose runtime results in VyPR processing data from many instruments in a small space of time (ie, systems with a high *event rate*).

Representing paths as flattened sequences of path condition IDs allows the set of potentially hundreds of queries for a single dynamic run to be reduced to one. This faster approach means that we can access the data that we need to analyse much sooner after it was generated by a running program.

6.6 An Offline Analysis Library

With a description given of how the information obtained during monitoring with VyPR is stored, we now discuss the analysis library for Python (found at `http://github.com/pyvypr/VyPRAnalysis/`), since this can be used to write offline analysis scripts that are used to automatically give indications of root causes of violations.

6.6.1 An Object-Relational Mapping Variant

The analysis library was introduced principally to allow querying of data obtained by monitoring of programs without in-depth knowledge of the relational schema. The underlying architecture of the library is an *object-relational mapping* (ORM), meaning there is a Python class for each table in the relational schema whose instance variables correspond to the columns in the table.

There are two key differences between the analysis library and a fully fledged ORM (such as SQLAlchemy [sql] in Python or CERN's CORAL [PCD+06] library in C++):

No need for update transactions. The analysis library only supports read operations, so much of the machinery implemented in an ORM to allow local changes to be mirrored by database update transactions does not need to be implemented.

No local database. The analysis library does not connect directly to the verdict database, rather to the verdict server which provides an API for querying the database. Hence, there is no need for database connection machinery (tools such as SQLAlchemy have substantial internal machinery to ensure the same ORM code can be used across databases) in the analysis library itself, and the database queries that take place are on the verdict server where we can be certain of the type of database being used.

We make a final remark that large verdict databases (for example those generated by the experiments discussed in Chapter 7) tend be in the order of tens of megabytes, meaning the memory requirements of the verdict server that queries such a database are not challenging to meet.

6.6.2 Queries with Post-Processing

Much of the analysis that one can perform on the data obtained during runtime by VyPR requires some post-processing once it has been returned from the server. The most significant examples are path-comparison and the interprocedural analysis that the analysis library supports so far.

Path Comparison

We consider the case of path reconstruction and subsequent comparison to demonstrate the effort that would be required to perform path-based explanation without the analysis library. A more in-depth view of the steps taken by the analysis library can be obtained by inspecting its source code, available at `http://github.com/pyvypr/VyPRAnalysis/`.

For this example, we assume that we have collected multiple dynamic runs from the same function in a system and that we would like to compare the program paths taken by each dynamic run to reach a certain instruction in the code. In the implementation terminology, we consider all paths leading to the same *instrumentation point*. The steps are the following:

1. For a given concrete state, we obtain the path condition ID sequence associated with its dynamic run.

2. We construct the symbolic control-flow graph over which we are interested in reconstructing paths.

3. Using the sequence of path conditions we obtained from the server, we apply Algorithm 8 over the symbolic control-flow graph to obtain a sequence of edges.

We remark that Algorithm 8 runs with a sequence of symbolic states rather than path conditions. However, given the sequence of path conditions, one can easily obtain the symbolic state found immediately after each path condition was satisfied, which enables one to apply Algorithm 8 given a sequence of path conditions.

After performing this reconstruction for multiple concrete states mapping to the same instrumentation point, based on our approach presented in Chapter 5, we have to:

5. Construct the context-free grammar for the symbolic control-flow graph.
6. Derive a parse tree with respect to this context-free grammar for each path that we reconstructed.
7. Intersect the parse trees.
8. Read off the leaves of the parse trees to determine path parameters.

The analysis library provides functions for every step integrated with the ORM, so the engineer can 1) choose to write their own analysis code if more flexibility is needed or 2) use higher-level functions that are provided to avoid thinking about the specific details if less flexibility is needed.

For example, if an engineer wanted to reconstruct the path leading to a concrete state (or *observation* in the less mathematical terminology used in the analysis library) which was stored in the variable `observation`, they would be able to write

```
observation.reconstruct_reaching_path(scfg)
```

to obtain a list of edges through the symbolic control-flow graph `scfg`. The computation performed by the method `reconstruct_reaching_path(scfg)` defined on an observation object consists of obtaining the path condition chain and then performing reconstruction.

A Remark on Implementation

The presentation of path reconstruction and comparison in this thesis relies on the fact that symbolic states are distinguished by *program points* taken from the program under scrutiny. The inclusion of these program points avoids symbolic states generated by similar statements in code being equivalent (since symbolic states only capture the potential for variable changes, two assignments to the same variable would otherwise generate the same symbolic state). In implementation, these program points are replaced by addresses in memory. For example, if two statements are similar, their symbolic states will be distinguishable because each will have a different address in memory.

This address-based distinction is exploited in the implementations of the path reconstruction and comparison algorithms. For example, in order to compare (in our case, intersect) two paths, equivalence checks are required to determine which symbolic states in one path correspond to which symbolic states in another. Therefore, in implementation, these equivalence checks are performed based on addresses.

A direct result of this is that, given that symbolic control-flow graphs are implemented using classes and objects, if two paths go through the same symbolic control-flow graph, they must

refer to the same symbolic states in the same *instance* of the symbolic control-flow graph. Hence, the relevant symbolic control-flow graph must always be a *parameter* given to any analysis being performed. This means that the symbolic control-flow graph must be constructed separately, and cannot be constructed as part of the path reconstruction/comparison process.

Interprocedural Analysis

VyPR collects enough information at runtime to determine the call tree that includes any function for which a specification was written. The main facility provided for this kind of analysis is given in the `VyPRAnalysis.interprocedural` module. In particular, in this module the class `CallTree` takes as input a transaction (HTTP request or program run), queries the verdict server for all calls taking place during it and then constructs the tree of calls, where a call is a child of another if and only if it occurred during the other. With a specific call object (obtained using other functions provided by the analysis library), the software engineer can then determine the calls below in the call stack using:

```
tree = VyPRAnalysis.interprocedural.CallTree(transaction)
callees = tree.get_direct_callees(call)
```

Further, if the path information exists, the paths of those calls below in the call stack can be reconstructed and compared across multiple runs. These paths can be obtained, assuming a common symbolic control-flow graph stored in the variable `scfg`, by writing:

```
paths_taken = map(lambda callee : callee.reconstruct_path(scfg), callees)
```

thus using Python's functional *map* facility to apply path reconstruction to each member of `callees`.

Interprocedural Analysis with Multiple Machines

If a software engineer wishes to analyse how the behaviour of a program on one machine affects the behaviour of another, one possibility is to synchronise the clocks on both machines and use VyPR to monitor specifications.

This application of VyPR is particularly interesting in the case in which a function on a client machine is responsible for requesting something from a server. Here, the behaviour that we can measure on the server has clear implications for what we measure on the client machine.

In practice, VyPR allows the software engineer to specify a server that provides the time via the Network Time Protocol (NTP). If this is done for both machines on which we monitor a program, then the situation we have is similar to that of interprocedural analysis; with a universal clock, we can reconstruct a call tree but where the calls are potentially generated by multiple machines. This technique is used in the experiment discussed in Section 7.7.2 to measure network stability.

For this technique to be possible, the required modification to the call tree construction process is trivial. To see this, we consider a system composed of a client and a server which communicate and we acknowledge that an HTTP request forms a nested transaction inside the existing transaction representing the client's runtime. Hence, during the construction of a call tree, we simply check for the existence of transactions whenever a call is found to have no calls

below in the call tree (hence is a leaf). If this is the case, we look for calls that took place on a different machine in order to fulfill a possible HTTP request sent by the call that formed a leaf of the call tree. In practice, we compute the call tree from the server and attach its root to the leaf of the call tree from the client.

One of the most important things that we can measure by synchronising multiple instances of VyPR across machines is overhead introduced by a network. For example, consider the configuration of machines and specifications in which we place a constraint over a function call (ie, a transition in a dynamic run) on machine A whose execution involves communication with B. Then satisfaction of constraints on machine B could lead us to conclude that network latency was to blame if the constraints on machine A were not satisfied.

Finally, by applying path comparison to this multi-machine setting, we can determine problematic branches on one machine that cause fulfillment of a request from the client to take much longer than expected. In general, we can extend our explanation approaches based on path comparison in the interprocedural setting to the inter-machine setting.

6.7 Explanation as a Tool

The ultimate goal of VyPR is to provide infrastructure with which an engineer's only interaction is 1) the initial writing of specifications; and 2) an inspection of problematic code based on verdicts obtained by monitoring for the specifications. This goal requires automation of instrumentation, monitoring and offline analysis. So far, we have given automatic methods for instrumentation and monitoring; it remains to *come full circle* and add a final piece to offline analysis to make it automatic.

Our approach is to implement explanation as a tool in multiple places in our ecosystem. Across all of these places, we identify a discrete set of cases to which we can apply our machinery and implement interfaces between the engineer and the underlying machinery to deal with those cases. Notably, we identify common ways in which specifications needed in real-world code bases can be violated and deal with those cases.

6.7.1 The General Intuition

In all cases of applying our explanation approach, we begin by identifying the atom, or atoms, that generated a failure for a given binding derived from a dynamic run. To determine such a set of atoms, we consider the bindings derived from the dynamic run in separation. The intuition here is that, for a program to yield false when monitored for satisfaction of a universally-quantified specification, failure requires *at least one* binding generated by the quantifier to give false. Hence, in our analyses we refer to this set of bindings (called *verdicts* in the terminology used in the analysis library).

Once we have the atom that caused a false verdict to be reached, which we call a *failing atom*, we must decide whether it is *normal* or *mixed* (see Chapter 3). This fact affects our explanation procedure.

6.7.2 Explanation for Normal Atoms

If an atom identified as failing is a normal atom, we apply a fixed set of analyses depending on whether the quantity required to decide a truth value for the atom was a transition or a concrete state.

Transitions

Consider the specification in (6.1). The atom that must cause violation in the specification is $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{check}))) \in (0, 0.2)$; a normal atom. This fact follows from the structure of the truth table of implication.

$$\forall q \in \mathsf{changes}(\texttt{db}) : q(\texttt{db}) = \texttt{"sqlite"} \implies \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{check}))) \in (0, 0.2) \tag{6.1}$$

In this case, the analyses we could perform include:

- Comparing paths leading to transitions used to evaluate $\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{check}))) \in (0, 0.2)$ that correspond to the same edge.

  This is likely to be most useful if the arguments given to the function `check` are computed by instructions in the control-flow leading up to the function call over which we have placed a constraint. In this case one could inspect the data-flow present (and indeed VyPR stores enough information to be able to perform such analysis without modification to the existing infrastructure).

- Reconstructing the call tree and comparing paths taken by functions below in the call stack.

  This is likely to be most useful if one cares little about the computation taking place on the path leading up to the function call, but does care about the call tree beneath it. In this case, VyPR can easily be told to record paths of function calls below in the call tree. Further, since path information is normally attached to the results of monitoring for specifications, VyPR can be told to record path information independently of monitoring for satisfaction of properties. Further, these paths are stored in the same way as the paths leading to observations, so reconstruction and comparison are straightforward and can be achieved using our existing approach.

  Finally, we observe that, since call trees can be reconstructed across multiple machines, this explanation approach can be extended to automate the detection of problematic regions of code on another machine with which a client communicates.

Concrete States

If we consider the specification $\forall q \in \mathsf{changes}(x) : q(x) \in (0, 1000)$ that requires that, whenever x changes, it remains strictly below 1000. In this case, it would be reasonable to:

- Reconstruct the path from the start of the function call to the concrete state at which x was given its new value.

- Reconstruct the path between successive changes of x.

This thesis will not address test cases that require these analyses, but we include this section for completeness and to demonstrate the generality of the explanation framework provided by VyPR.

6.7.3 Explanation for Mixed Atoms

If the atom that led to failure for a given binding is mixed (recall that this means that the atom requires multiple observations to be resolved into a truth value, described more in Chapter 3), the types of analysis we can perform differ from the normal atom case. The types of analysis that we perform if the atom that generates a failure is mixed also depend on more attributes of the atom than just its status as either *normal* or *mixed*. This is because declaring an atom as *mixed* does not say a lot about the type of constraint it places. For example, consider the following specifications, one new and one repeated from before but displayed here for completeness:

$$\forall c \in \mathsf{calls}(\texttt{construct}) : \mathsf{timeBetween}(\mathsf{dest}(c), \mathsf{dest}(\mathsf{next}(c, \mathsf{calls}(\texttt{commit})))) < 0.5 \tag{6.2a}$$

$$\forall q \in \mathsf{changes}(x) : q(x) < \mathsf{next}(q, \mathsf{changes}(y))(y) \times 1000 \tag{6.2b}$$

Time Between States

For the property in (6.2a), for each binding derived from the dynamic run that is being checked, we have two concrete states that must be processed to determine whether the constraint holds. If a mixed atom uses the timeBetween operator, we do not care about the values held by the concrete states used to determine the necessary time information. Therefore, a natural first step is to record the path between the two concrete states and perform comparison. There are two analyses that we currently consider:

- Compare branching along paths between concrete states that were all generated by the same pair of instrumentation points.

  To perform comparison of paths, we always require that all paths in the set we consider start and end at the same symbolic states in the symbolic control-flow graph. This prevents, for example, comparison of paths from completely different parts of the symbolic control-flow graph. This constraint is met if we determine the pairs of instrumentation points associated with every pair of concrete states used by the atom. Hence, our approach is to group paths by pairs of instrumentation points, then group by the truth values generated and ultimately try to determine whether a certain feature is often (but not necessarily always) present in failing paths. Such a feature could be a branch taken through a conditional block or the relationship between time taken and the number of iterations completed by a loop.

- If there is no difference in branch taken, compare timing along paths between concrete states.

If the comparison of paths using our existing technique does not show anything useful, then a reasonable next step is to compare the timing information. This can be obtained by additional instrumentation by VyPR. Further, if a set of paths is found to be the same (there are no path parameters), we can compare the timing information attached to the symbolic states from which there were multiple possible paths to take.

Values Between States

For the property in (6.2b), we again require concrete states but this time we care about the values present in either state. Given two concrete states required to decide a truth value of such a mixed atom, assuming their timestamps are not equal, there is necessarily a total order. In this case, we are interested in comparing the paths taken between the concrete states in the same way as for the timeBetween case.

Considering the atom $q(x) < \mathsf{next}(q, \mathsf{changes}(y))(y) \times 1000$, we would be interested in the computation that took place to mean that the value of y recorded did not satisfy the constraint. In general we can say that, of the concrete states required to determine a truth value of the atom, the only one that can cause violation is the one that is maximal; it is obliged to satisfy the constraint given the value contributed by the one that is minimal.

6.8 Web-based Analysis Environment

The same service that provides a repository for storage of verdict data and an API for their access also provides a graphical analysis environment. This tool removes the need to write analysis scripts for some common use cases, allowing software engineers to navigate and visualise the data generated by VyPR without writing any code.

Like most web applications, the analysis environment is implemented in two main components: the front-end and back-end. The back-end is a collection of end-points that serve 1) components of the analysis environment's GUI (front-end) and 2) an API for the front-end to get the data it needs in a useful format.

6.8.1 Usage of the Analysis Environment

The front-end (user-facing part) of the analysis environment is written in Javascript with help from the Vue framework [vue]. Figures 6.1, 6.2, 6.3, 6.4 and 6.5 show screenshots of the environment at key stages.

Specification Listing. Figure 6.1 shows how specifications are listed for engineers to select them, with respect to the structure of the monitored system. Selecting a specification gives a list of calls of the relevant function, along with the duration and verdict generated by each call.

Interacting with a Specification. Figure 6.2 shows that, once a specification and a set of relevant function calls have been selected, the specification itself is interactive. In particular, once a part of the code is chosen that was identified by a quantifier from the specification, the subsequent selection of a part of the specification will highlight the parts of code identified by

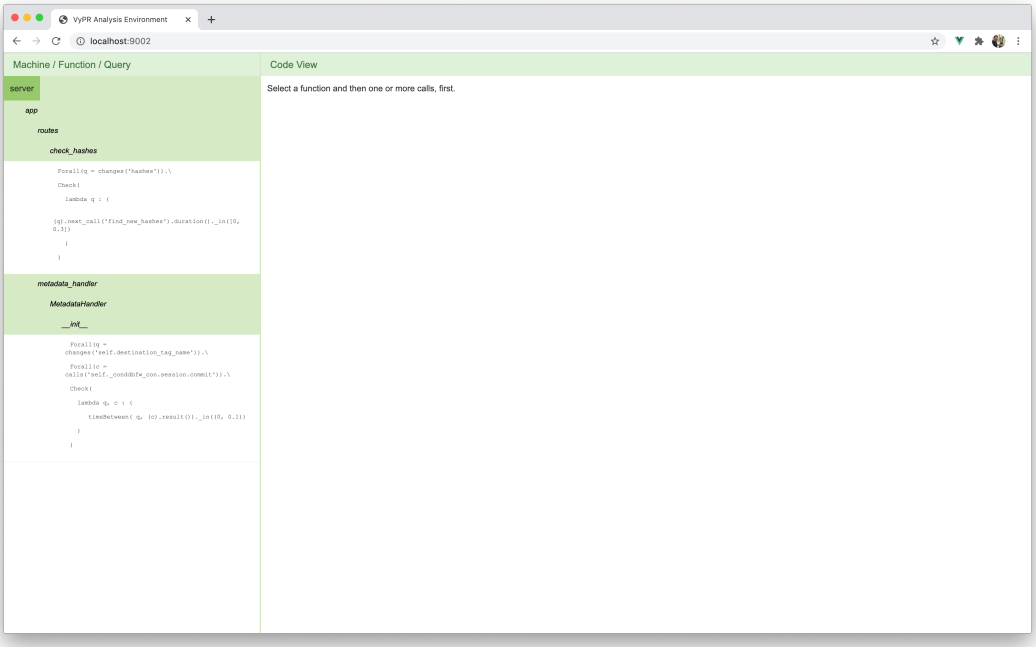


Figure 6.1: The interface provided by the analysis environment for selecting specifications written by engineers.

instrumentation. The selection procedures changes depending on the form of the specification. For example, the specification in the screenshot includes a mixed atom using the timeBetween operator, meaning that the engineer must select two statements in code that were instrumented to see the relevant measurements.

Plotting Observations. Figure 6.3 shows how the observations recorded by VyPR can be plotted in order to see the raw data used to decide a verdict for the selected specification.

Triggering the Explanation Machinery. Figure 6.4 shows how the analysis environment uses the path-based explanation approach described in Chapter 5 in order to highlight for engineers which points in their code may lead to degraded performance. The window opened once the engineer decides to plot performance data according to the path taken is shown in Figure 6.5.

Ultimately, the analysis environment allows software engineers to inspect performance query results obtained by VyPR monitoring in context with the source code itself. This is similar to existing work on visualising performance data [CLRG19, BMDR13, MMRL16, COL+17, RHV+09], the key difference being that our performance data is with respect to formal specifications written in CFTL, rather than a conventional profiling approach. Further, unlike existing work, we do not yet display performance data inline with code in an Integrated Development

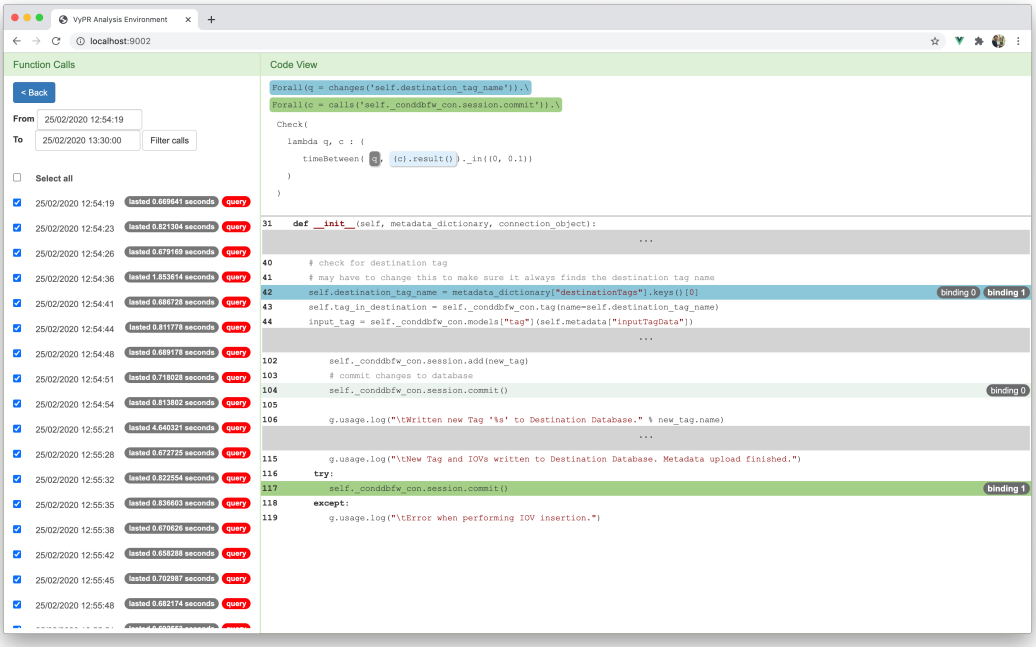


Figure 6.2: The link between components of a specification, and the source code identified by performing instrumentation for those components.

Environment (IDE). Rather we display it inline with code that we read from a source file without any ability to make changes. In general, addressing the IDE case was out of the scope of this work.

Finally, the low-level nature of CFTL allows us to easily link results from monitoring to the relevant source code. For example, given a specification we enable engineers to see which parts of their code were relevant to parts of their specification. This allows detailed analysis of what happened at runtime with respect to the specifications written over the system.

6.9 First Steps

This chapter has presented the implementation that forms the first steps in applying Runtime Verification to automate performance analysis of Python programs with respect to low-level specifications. The VyPR framework allows engineers to write *queries* over their programs and then gives two options: write scripts using the library distributed alongside VyPR, or use the powerful web-based analysis environment. The analysis facilities developed have taken some initial steps towards being able to identify root causes of violations of low-level properties in the interprocedural and inter-machine setting.

In the next chapter we will describe application of our implementation to software used at CERN. In particular, VyPR was used as part of the development process to investigate performance in environments provided by CERN infrastructure. Further, VyPR was put under realistic loads and what we report in the following chapter was obtained by running VyPR in

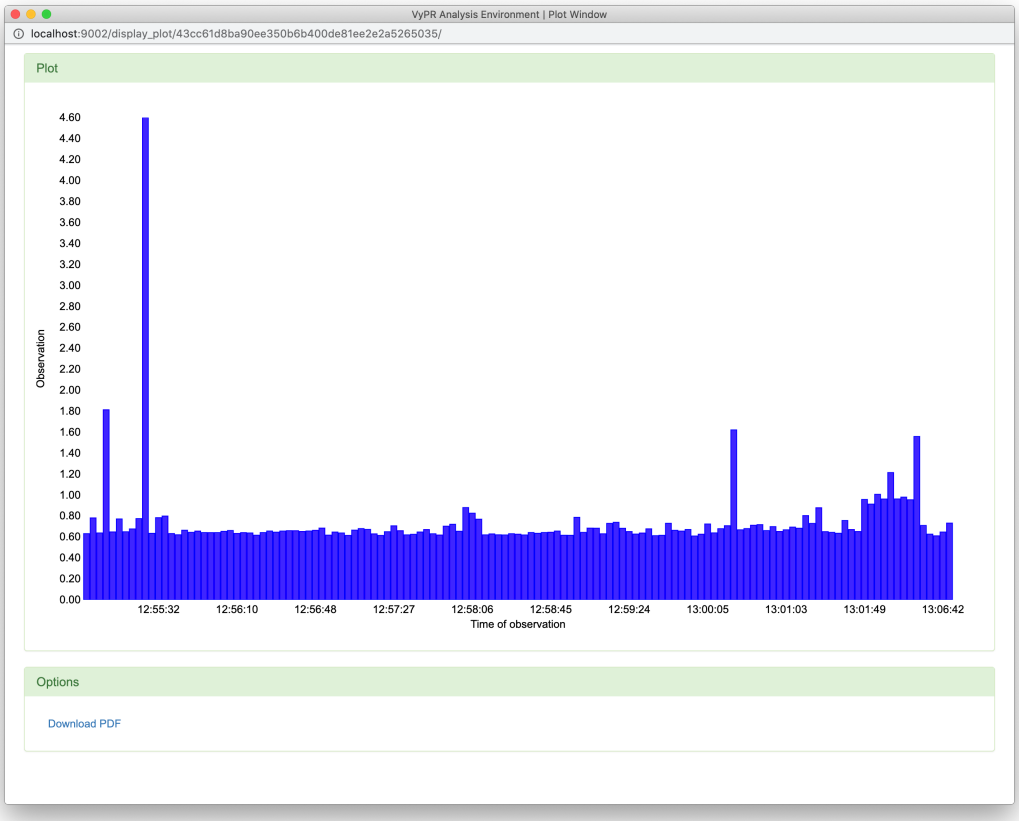


Figure 6.3: An example of plotting measured timing information based on a single statement in code.

environments that were as close to production as possible (including similar virtual machine infrastructure, database systems and surrounding networks).

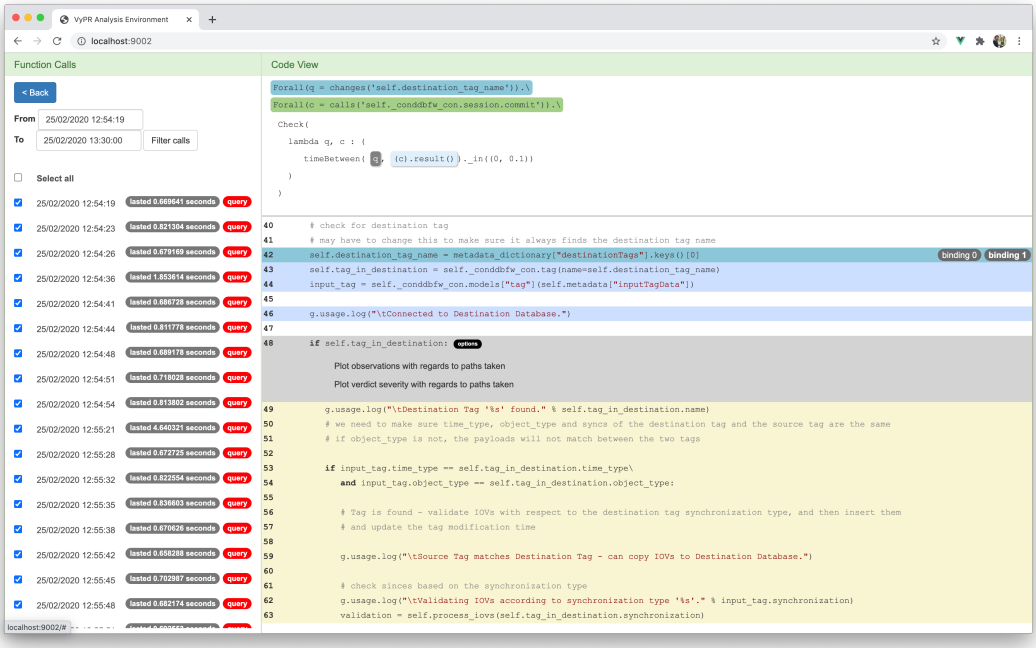


Figure 6.4: An example of detected divergence of paths, and the enabled user interaction.

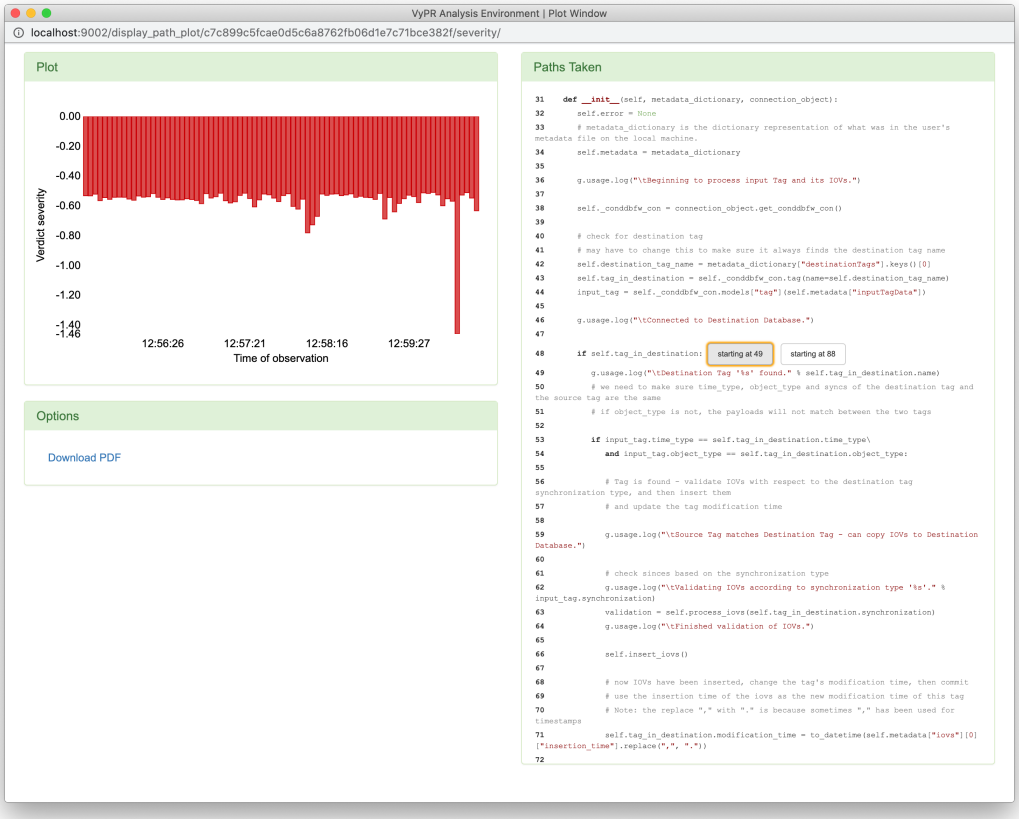


Figure 6.5: An example of plotting measured timing with respect to a single path taken through code.

Chapter 7

Application of VyPR

This chapter will describe the application of VyPR to real software used at the European Organisation for Nuclear and Particle Physics Research (CERN) in Geneva, Switzerland. The chapter will be based partly on the material in the papers presented at TACAS 19 [DRF+19] and CHEP 19 [DHR+19]. The TACAS 19 paper describes the first application of a prototype of VyPR to a web service whose analysis plays a large role in this chapter. The CHEP 19 paper describes further investigation into this service. In addition to the material presented in these papers, this chapter will present further analysis of the same web service using more sophisticated machinery developed since their publication.

7.1 Opening Remarks

All of the experiments presented were performed in environments that were as close as possible to the relevant production ones. This involved using the same CERN OpenStack [cer] infrastructure to provide machines on which to run the programs, the same database services and the same surrounding network. Further, all analysis was performed using the facilities provided by VyPR, including:

- The central server, for storing and supplying information obtained during monitoring.
- The analysis library, for querying the central server and performing any necessary post-processing, especially for path comparison.

In cases where the post-processing of data was not straightforward, we describe the script that was written using VyPR's analysis library.

Further, all specifications given were written first and then reproduced (given natural language versions of the specifications) by the engineer making the changes to the monitored service. We highlight that specifications have been written by engineers on two projects to which VyPR has been applied. All offline analysis was performed by us since development of the analysis tools was done partly according to the use cases in this chapter.

7.2 Applications at the Compact Muon Solenoid

The Large Hadron Collider (LHC) [EB08] is a circular proton-proton collider that is part of CERN's accelerator complex. Around its circumference are found four main *interaction points*, where two proton beams circulating in opposite directions are made to collide. One such interaction point is instrumented with a complex omni-purpose detector facility, the Compact Muon Solenoid (CMS) [Col08].

During the operations of the LHC, the CMS experiment produces raw data which need to be processed in order to identify all particles with their position and kinematic properties, as needed for subsequent analysis. Such preprocessing combines *event* data, i.e. the raw information specific to collisions, with *non-event* data, which describe the slowly evolving status of the detector (like the accurate location of its sensitive volumes and the characteristic energy response). This preprocessing is key to exploiting the ultimate scientific performance of the CMS apparatus.

7.3 Upload of Alignment and Calibrations Constants

The *non-event* data taken during LHC runs is often referred to as *conditions* or *alignment and calibrations constants*. The service that we will analyse, called the Conditions Upload service, is responsible for sending these alignment and calibrations constants to a central database. Once uploaded there, the constants are used in processing of the raw data. In the remainder of this chapter, we will refer to the data processing as *reconstruction*, reflecting the fact that its outputs are C++ physics objects. Such physics objects are candidates to represent particles which emerged from the collisions, and are the typical inputs to the data analysis effort by members of the CMS Collaboration.

The service with which we concern ourselves is currently undergoing final work before the transition into production for Run 3 of the LHC in 2022 and VyPR is being used to ensure a performant final product. The final work, and any further maintenance during LHC runs, includes:

- Addition of logic to handle new use cases.
- Ensuring that the checks carried out by the service on data proposed for inclusion in the central database are correct. As the code base changes to handle new use cases and changing structure of Conditions data, maintaining correctness is important.
- Ensuring the service remains performant.

Whichever method is chosen to get assurance that the service remains performant, its application must be straightforward and results must be accessible quickly. That is to say that the development process should not be perturbed by the application of any analysis technique. The priority is to deploy code that satisfies the physics use cases without investing a lot of time in analysis, while being able to easily act on the feedback from such analysis.

7.3.1 The Structure of Conditions Metadata

The Conditions data used in reconstruction by CMS must define 1) the alignment and calibrations constants associated with a particular sub-detector of CMS and 2) the time interval during which those constants are valid. These alignment and calibrations constants are represented by *Payloads*. The intervals during which each Payload applies is defined by an *Interval of Validity* (IOV). Since alignment and calibrations are specific to individual parts (referred to formally as *sub-detectors*) of CMS, IOVs are collected together in *Tags*. Finally, some types of Conditions must satisfy constraints set by other machines in CERN's network, while others are not constrained at all.

For example, the tag `BeamSpotObjects_LumiBased_v4_offline` contains information regarding the region at the centre of the CMS detector where the counter-rotating proton bunches collide, generating the secondary particles which are detected by the experiment. By browsing the IOVs in this Tag, one can find the relevant constants for a given point in time.

7.3.2 The Conditions Upload Process

The Conditions Upload service is responsible for ensuring the IOVs are constructed correctly in the central database and the Payloads to which those IOVs refer are present. Errors in this process could lead to incorrect Conditions and, ultimately, inaccurate reconstruction of physics objects. Once correctness is established, it is also vital that we understand the timing behaviour exhibited. The reasons for this understanding are presented later, since they depend on the part of the Conditions Upload service being analysed.

Before describing our analysis approach, we describe the process employed to upload Alignment and Calibrations constants to a central database. The process is designed to 1) deal with unstable network connections when uploading large amounts of data, and 2) ensure that correct validations are performed, especially when multiple clients around the world are involved. The steps of the upload process can be enumerated as such:

1. Open a new upload session with respect to a Tag. Only one user at a time can upload to a Tag.
2. Ask the server which of the Payloads proposed for upload already exist. We only send the Payloads that the server tells us it doesn't have.
3. Send each Payload in a separate HTTP request.
4. Once the Payloads are uploaded, send the IOV and Tag (collectively referred to as *metadata*) information.
5. Once all of the metadata has been sent, the upload session is closed so other users can upload to the Tag.

This process employs significant optimisations to reduce the amount of data being sent. For example, in some cases data being sent already exists in the central database. We detect instances like this by means of the client communicating with the server, thus reducing the work done by both.

7.3.3 Distinguishing Architectural Characteristics

The Conditions Upload service uses a client-server architecture. While it is ultimately a web service, there is also a lot of computation performed by a client program. The client is used by 1) physicists who need to add some Conditions to the central database; and 2) CERN's Tier0 [tie] for upload of Conditions after automated computation during LHC runs. In particular, during LHC runs, Tier0 performs automated computation of Conditions ready for the translation of data recorded by the CMS detector into objects that can be used by physicists during analyses. The Conditions Upload service is used once the automated computation has been completed.

Upload is performed by reading in locally-stored database and metadata files, performing some checks by sending requests to certain end-points provided by the server and ultimately uploading the appropriate parts of the locally stored database.

Based on this architecture, it is reasonable for maintainers of the Conditions Upload service to want to perform analysis of the behaviour of both the client and the server-side programs. In particular, the performance of the server-side affects the client-side directly; a delay in some operation performed on the server is reflected in a delay to the response received by the client. This fact should be reflected in our analysis methods.

To allow analysis of the interaction between the two machines, we apply VyPR to both the central upload service and the client program from which the Conditions are sent. We also point both instances of VyPR to an NTP server in order to synchronise their clocks. We prefer the same clock to increase the chances of accurate synchronisation.

Aggregation from Distributed Monitors

Ensuring that both instances of VyPR are pointed at the same verdict server, as discussed previously, we can extend our analysis techniques from the interprocedural setting to the inter-machine setting.

In the case of the Conditions Upload service, we apply VyPR to the client-side and server-side code bases, synchronised with the same clock, and collect data from both during uploads. When we write the specification for each code base (specifications cannot express properties concerning both code bases at the same time), we ensure that functions that we monitor on the client-side interact via HTTP with the functions we monitor on the server-side. This way, any analysis we perform can link results from each code base.

7.4 Analysing the Conditions Upload Service

We now discuss the parts of this process that we are interested in monitoring with VyPR. We will address two main categories of problems:

- When detector experts perform uploads, operational experience of the Conditions Upload service has shown that there can be problems with large amounts of data being sent over a perhaps unreliable network. In which environments, and for which data, is this likely to occur?

This is important from the perspective of assigning each client (in an environment with possibly multiple clients) an appropriate amount of time to perform their upload. If we adopt a policy of giving a fixed amount of time to a client before their upload must be stopped, we need to be sure that realistic upload sizes are allowed and that factors such as erratic latency from network instability are taken into account.

- Do the optimisations we use have enough effect on the overall upload time to be worth performing for every upload? Are there certain types of uploads for which we can avoid the check?

  Existing optimisations, the most predominant one being a way to avoid sending Payloads unnecessarily, have been demonstrated to reduce upload times by orders of magnitude [Daw17]. However, there is no evidence yet that the effect is the same for all uploads and deciding whether the optimisation is worth applying in every case requires further analysis.

We now describe the precise parts of the upload process that we would like to investigate. Our description of each part will include its significance to the upload process (including why an insufficient understanding of its behaviour could be problematic) along with how we use VyPR.

7.4.1 Hash Checking

Hash checking is an optimisation applied before sending all Payloads to the server-side. Since Payloads can be large (around 100 MB or more) and there can be many in a single upload, we would like to reduce the number that we send by asking the server which Payloads are already stored. We describe our analysis of this part of the Conditions Upload service's code in Section 7.7.1.

In the case of this optimisation, we ask two questions. First, we want to know how the time taken to perform the check scales with the number of Payloads being checked. Second, we want to know in which cases spending the extra time to perform the check reduces the number of Payloads being sent significantly enough to justify the time taken by the optimisation. This will depend on the types of Conditions being uploaded, since management of Conditions varies between users (some may always send new Payloads, and others may have the habit of using existing ones).

To determine how the time taken scales with the number of Payloads checked by the optimisation, on the server-side we use the CFTL specification

$$\forall q \in \mathsf{changes}(\texttt{hashes}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{check\_hashes}))) \in [0, 0.3] \tag{7.1}$$

This specification was defined after some iteration on values and gives an intuitive *reasonable amount of time*.

To answer our second question about how many Payloads are thrown away by this optimisation, we write the CFTL specification

$$\forall c \in \mathsf{calls}(\texttt{find\_new\_hashes}) : \mathsf{source}(c)(\texttt{hashes}) \times 0.7 < \mathsf{dest}(c)(\texttt{not\_found})$$

to express the requirement that hash checking should determine that at least 70% of proposed hashes were not found. We chose 70% as an initial value to investigate how often hash checking reduces the data to be uploaded significantly, while acknowledging that sending no data is likely to be quite rare. Section 7.7.1 gives the results of an investigation using this initial value which did not give rise to the need to refine the value.

Analysing the violations of this specification gives us an idea of how many uploads were proposing existing Payloads versus how many were proposing new ones. This helps us in determining whether certain categories of Conditions uploads do not need the hash checking optimisation.

7.4.2 Payload Upload

Each upload includes metadata which specifies a local database file. Uploading Payloads over HTTP is a matter of sending a POST request to the server from the client once such Payloads have been read in from the local database file. We describe our analysis of this part of the Conditions Upload service's code in Section 7.7.2. In particular, we will concentrate on the time taken to send a Payload over HTTP and then to perform the insertion into the central database. Hence, we care about the behaviour of two connections separately. We cannot take the measurements that we need just by placing specifications over the server-side; we need measurements from both the client and server-sides to detect problems in both connections. On the server side, we write the specification

$$\forall q \in \mathsf{changes}(\texttt{payload}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{insert}))) < \mathsf{length}(q(\texttt{payload})) \times k$$

where k is a positive real number much less than 1 with units *seconds*. We include the constraint that k should be much less than 1 in order to capture the intuition that the call of `insert` should not take a number of seconds equal to the size of the Payload (ie, if length gives the size of the Payload in bytes, a single byte should take far less than a second to be written to a database). Hence, this specification requires that the upload time is bounded by some (small) fraction of the size of the Payload.

We apply the same specification on the client-side, possibly with an adjusted constant k. By monitoring for satisfaction of this specification on the client and server-sides, we can measure the network behaviour. This follows from the fact that if the specification is violated on the client but not on the server, then we can conclude that the problem is likely to be an unstable network connection. In this case, while we cannot do anything with the code itself to improve the situation, we can eventually see if there are times at which we can anticipate the network to be erratic.

Finally, using the facilities provided by VyPR, these problems can be detected automatically. In particular, if there is information from monitoring on both the client and server-sides,

VyPR can construct the call tree consisting of call information from both machines and use failures of callees to explain failures in callers.

7.4.3 IOV Uploads

IOVs are inserted after Payloads are uploaded (since IOVs require the existence of Payloads). Since the validation process for IOVs is different depending on whether or not the target Tag of the upload already exists, we investigate the time taken by the validation process in either case. We detail our analysis in Section 7.7.3.

In the source code of the Conditions Upload service, these two cases are represented by distinct paths through the same function. Hence, we choose to measure the time taken to get between two points and compare the paths taken. This way, we can determine whether one branch (if a Tag is new or not) is more likely to cause IOV validation to take an unacceptably larger amount of time. We use the specification

$$\begin{aligned} &\forall q \in \mathsf{changes}(\texttt{tag}) : \\ &\forall t \in \mathsf{future}(q, \mathsf{calls}(\texttt{commit})) : \\ &\qquad \mathsf{timeBetween}(q, \mathsf{result}(t)) < k \end{aligned}$$

where k is a positive real number with units *seconds*. Using VyPR's analysis scripts, we can compare the paths taken from q to t and determine if either of the two possible paths is problematic. Notice that we have to universally quantify over calls to `commit` (a function that performs a database commit in the Conditions Upload service's source code), rather than using the next operator. This is because, on one branch in the source code, there is a call to `commit` before the one we care about.

7.4.4 Access Control

At the beginning of any request handling that the Conditions Upload service performs, there is an access check implemented by the class `Usage`. Instantiation of this class allows checking that the upload being performed will not disrupt any other users who obtained permissions to upload to a given Tag first. For this analysis, we use multiple specifications, but the weight of the work is done in post-processing. Section 7.7.4 gives more detail.

7.5 Experimental Setup

In order for any experiments that we ran to be as close to the production setting as possible, we recorded Conditions uploads from the service that is currently in production over a period of 2 years. This resulted in a repository of approximately 35,000 Conditions uploads, though we have not performed experiments using the entirety of this repository because a subset suffices.

The machines we used in our experiments were CERN OpenStack virtual machines. For the central Conditions database, we used the same type of database (Oracle) as that used in

production, but from a different cluster. For experiments involving a client machine uploading to a server, we use separate OpenStack virtual machines.

7.5.1 Upload Replay

The process which begins with a user asking the Conditions Upload server for a session in which they can perform an upload, and finishes with the user being informed that their upload has succeeded/failed, is known as an *upload*. All experiments performed in this chapter are done so using *replays*, which are executions of sets of uploads. Replays are normally of all uploads starting from the first one ever recorded going up to some limit.

It was important to simulate the contents of the central Conditions database since a lot of the computation performed depends on the state of this database. With this in mind, before every replay we deleted the entire contents of the database and reconstructed a snapshot of it using the timestamp of the first upload used in the replay. To save time, we did not copy entire Payloads (since these can be more than 100MB each and there can be tens of thousands) from the snapshot, rather we inserted fake data (a few bytes). While we expected this to have no effect on the performance being measured, the preliminary investigation described in Section 7.6 gave signs of the contrary. However, further investigations described in Section 7.7.1 showed that the conclusions that are presented do not depend on this approximation. Further, the fix applied in the same section removes dependence of any analysis on this approximation.

7.6 Initial Experiments

We describe the initial experiments that we performed with a prototype of VyPR on the Conditions Upload service, which were described in the TACAS 19 paper [DRF+19]. These experiments were performed before any explanation machinery had been developed; VyPR's explanation facilities were developed with this service as a use case.

Our initial work on the Conditions Upload service was split into two experiments. These experiments used real upload data from LHC runs to make the experiments as close as possible to the reality of Conditions being uploaded during LHC runs. We used CERN OpenStack [cer] infrastructure (in agreement with the production setting) for the client and server-sides and an Oracle database backend. Our experiments included:

- A full replay of 6 months of Conditions uploads collected during Run 2 of the LHC.
- A replay of a subset of the 6 month dataset to check Payload insertion latency.

We first describe the experiment involving full replay of 6 months (approximately 14,500 Conditions uploads) of data. Our specification was as listed below. We include the location in the code base in Python-style module notation (for example, app.routes.check_hashes refers to the function `check_hashes` in the module `routes` in the package `app`).

Property ID 1 app.usage.Usage.new_upload_session

$$\forall q \in \mathsf{changes}(\texttt{authenticated}) : \forall t \in \mathsf{future}(q, \mathsf{calls}(\texttt{execute})) : \left(\begin{array}{c} q(\texttt{authenticated}) = \text{True} \\ \implies \mathsf{duration}(t) \in [0,1] \end{array} \right)$$

Whenever **`authenticated`** *is changed, if it is set to* **`True`**, *then all future calls to* **`execute`** *should take no more than 1 second.*

Property ID 2 app.routes.check_hashes

$$\forall q \in \mathsf{changes}(\texttt{hashes}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{find\_new\_hashes}))) \in [0, 0.3]$$

When the variable **`hashes`** *is assigned, the next call to* **`find_new_hashes`** *should take no more than 0.3 seconds.*

Property ID 3 app.routes.store_blobs

$$\forall t \in \mathsf{calls}(\texttt{con.execute}) : \mathsf{duration}(t) \in [0, 2]$$

Every call to the **`con.execute`** *method on the current database connection should take no more than 2 seconds.*

Property ID 4 app.metadata_handler.MetadataHandler.__init__

$$\forall t \in \mathsf{calls}(\texttt{insert\_iovs}) : \mathsf{duration}\left(\begin{array}{l} \mathsf{next}(t, \\ \mathsf{calls}(\texttt{commit})) \end{array} \right) \in [0, 1]$$

Every time the method **`insert_iovs`** *is called, the next commit after the insertion should take no more than 1 second.*

Property ID 5 app.routes.upload_metadata

$$\forall t \in \mathsf{calls}(\texttt{MetadataHandler}) : \mathsf{duration}(t) \in [0, 1]$$

Every time **`MetadataHandler`** *is instantiated, the instantiation should take no more than 1 second.*

The plots in Figures 7.1 and 7.2 show the number of violations generated by these properties. In Figure 7.1, we show the ID of each property (derived from the verdict database) with the number of violations observed during the replay for that property. Clearly, from this plot, the violations of the property with ID 2 on the server-side exceed those caused by other properties by an order of magnitude. This property is monitored over the hash checking optimisation, so this many violations makes us suspect that the optimisation is problematic. It is for this reason that we wonder how often the optimisation has a large enough effect to be included and, from this, whether it needs to be included in every upload. If there are many uploads for which the optimisation takes a long time and has little effect, it is natural to either 1) characterise them and possibly turn off the optimisation for others in the same category; or 2) make the optimisation optional for all uploads.

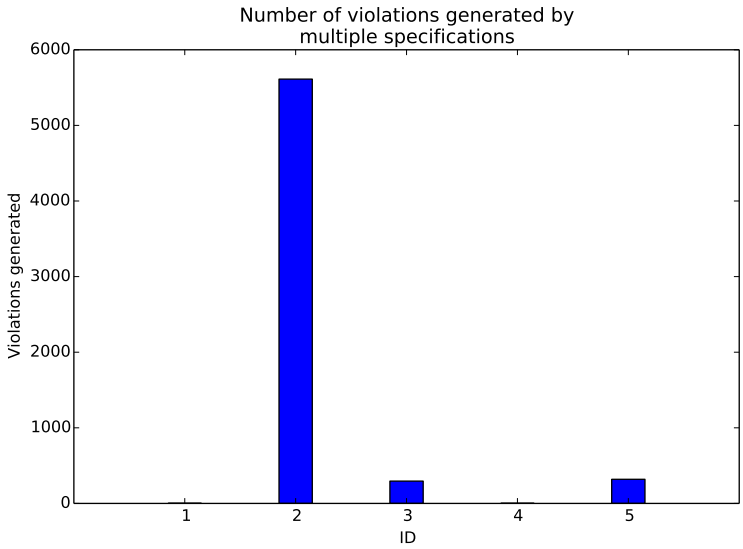


Figure 7.1: The number of violations of each specification, monitored over 14,610 uploads.

Figure 7.2 shows violations counted for the property with ID 3 in the list above on the function `store_blobs` across different replays of the same 900 uploads. Each replay involved a different delay between each upload. For each upload, this function is called multiple times since each upload can send multiple Payload blobs. One can see from this plot that uploads performed with less time elapsed since the previous can generate more violations. This behaviour is not due to VyPR's monitoring mechanism because the time taken for the Payload blob to be sent from the client to the server is enough time for the monitoring mechanism to perform any work that it needs to do. Given that the number of violations observed here is small compared to those seen in Figure 7.1, this behaviour is less of a priority to investigate, but it remains an issue nonetheless.

It is suspected that the underlying database connection machinery performs some operations that are only noticed when queries are performed in quick succession. Hence, a solution after further investigation may be to use the underlying database library differently.

We demonstrate the beginning of a long-term investigation into the behaviour presented by Figure 7.1 in Section 7.7 because this behaviour of Payload insertion (higher latency when insertions are performed closer together) is unexpected and problematic for high loads on the server.

7.7 Further Experiments

Based on the results of the experiments reported in the TACAS paper, we have investigated certain parts of the Conditions Upload service further. Our investigations will be used as feedback to the ongoing maintenance process and will help the service to be as performant as possible during the next LHC run. We remark that the results presented from now on were obtained some time after the first results, when loads on the surrounding network were less (due

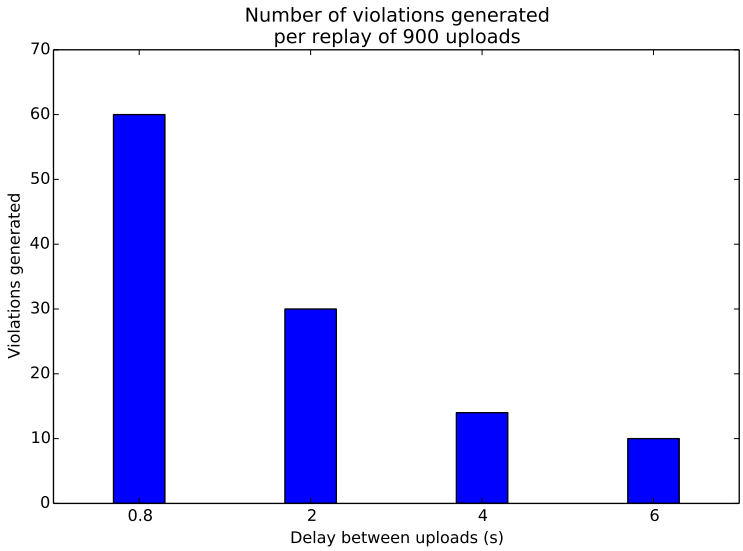


Figure 7.2: The number of violations of parts of the database query constraint vs replays of the 900 upload dataset. For each replay, the delay between uploads was different.

to reduced physics activity). Due to this, some statistics will change but, since we are working with the same Conditions Upload service code base as the first experiments, any performance issues will still be present.

7.7.1 Behaviour of Hash Checking

One of the most critical parts of Conditions Upload is the determination of which Payloads need to be sent to the central server, otherwise known as *hash checking*. If this process works smoothly, then the upload process is often greatly optimised. Otherwise, it can be the case that the optimisation takes more time than it justifies. In this section, we investigate its effectiveness in depth.

Time Taken by Hash Checking

In the cases where the time taken by hash checking is higher than expected, we need to determine whether network latency or number of hashes is mostly to blame. We begin with the specification we give to VyPR, which is

```
Forall(q = changes("hashes")).\
Check(
  lambda q : q.next_call("find_new_hashes").duration() <= 0.3
)
```

To investigate how the time taken varies with the number of hashes, we opt against just storing the list of hashes; some uploads contain in the order of thousands of hashes, and each one is 40 characters long. This would be an unnecessary amount of data to store.

VyPR's path and interprocedural analysis tools become useful here. This is the case because hash checking is currently performed by iterating over the list of hashes sent to the server and

checking the existence of each one. Hence, rather than storing the entire list of hashes, we can instead refer to the path information collected by VyPR. There is still the need to store additional information, but less than storing 40 character hashes (for small lists of hashes, the difference is small, but for lists containing 1000 or more hashes, storing ∼1000 integers rather than ∼1000 40 character strings over time, the difference is more noticeable). Ultimately, by storing the path information we demonstrate a general technique that can be applied in situations far worse than this one (where the alternative to path reconstruction may be storing data containing complex objects, rather than simple strings).

We perform this analysis using VyPR's analysis library. We assume that the library package is stored in the object `va`, then write a script that takes the following steps:

1. Get the function during which we measure the time taken by the hash checking optimisation. We do this using

   ```
   hash_checking_function = va.list_functions()[2]
   ```

   This implicitly fixes the property that we care about because properties are paired with functions in the version of the VyPR ecosystem with which the analyses in this chapter are carried out.

2. Obtain the list of calls of the function using

   ```
   all_calls = hash_checking_function.get_calls()
   ```

3. For each `call` in `all_calls`:

   (a) Get the single observation that we know took place during this call using

   ```
   observation = call.get_observations()[0]
   ```

   (b) Determine the verdict to which this observation contributed. We only proceed with this call if the verdict gave failure.

   ```
   verdict = va.verdict(id=observation.verdict)
   if verdict != 0:
     break
   ```

   (c) Determine the transaction during which `call` took place using

   ```
   transaction = va.transaction(id=call.trans)
   ```

   (d) Construct the call tree of that transaction using

   ```
   tree = CallTree(transaction)
   ```

   (e) Determine the child of `call` in the call tree. This will be the function that performs the optimisation. We use:

   ```
   child_call = tree.get_direct_callees(call)[0]
   ```

   (f) Get the path taken by that child call, assuming a symbolic control-flow graph stored in the variable `scfg`, using

```
path = child_call.reconstruct_path(scfg)
```

(g) Pair the time taken by the optimisation with the number of iterations completed by the loop that checks hashes.

We omit code in this case because it performs simple processing on the data obtained so far and does not make further use of the VyPR analysis library, so reveals no further insight.

4. Finally, group the pairs into bins defined by the number of hashes checked and generate a plot.

Plotting is not covered by the VyPR analysis library and, as such, is left up to the software engineer.

We now make some observations about this procedure. These lead to important optimisations, especially when analysing larger datasets.

First, *transactions*, in the web service setting, correspond to HTTP requests. The upload process consists of multiple HTTP requests being sent to the upload server. This means that we cannot simply iterate through transactions and determine the function call that corresponds to the hash checking optimisation; it is not present for every transaction. The solution in our analysis procedure is to determine the calls to the relevant function and, from there, determine the transactions.

Second, to count loop iterations we do not actually need to fully reconstruct the path taken through the relevant symbolic control-flow graph. Instead, we can simply count the number of entries in the path condition sequence that correspond to a loop body being entered. Since the optimisation function contains only a single loop, we can be sure that the path conditions we count are from the same loop.

Using this approach, Figure 7.3a shows the average time taken by the hash check for the same bins as the other figures. This plot was obtained by running a larger set of 14,000 Conditions uploads (containing the smaller dataset used for other experiments), so we represent the distribution of values for each bin. We see that the average time taken by hash checks fitting into the final bin is significantly higher than for the other bins.

Hash Checking Effectiveness

Next, we investigate the effectiveness of hash checking by writing a constraint, in effect, over the effectiveness itself. Our specification is the following:

```
Forall(q = calls("check_hashes")).\
Check(lambda q : (
    q.input()("hashes").length() * 0.7 < q.result()("not_found").length()
  )
)
```

which expresses the requirement that the number of hashes not found should exceed 70% of the original list of hashes. The intuition here is that the optimisation should have a sufficient effect on the upload process that its removal would cause significantly different behaviour. We

quantify over calls of `check_hashes` to be sure that the hash checking operation has taken place.

We show two plots obtained by analysing the results of this specification in Figures 7.3b and 7.3c, derived from running 4000 Conditions uploads. Figure 7.3b shows the average percentage of Payloads not found by uploads, presented in non-uniform bins. Figure 7.3c is similar with all data contributing to succeeding verdicts thrown away. Considering 7.3b and 7.3c together highlights an important observation: Conditions uploads which propose a high number of Payloads often upload Conditions that do not already exist. From this we take two key results:

- We must consider making hash checking an optional part of the Conditions upload process if it takes a long time in cases where it has little benefit.
- Instances where hash checking takes a long time are often those where the Payloads do not exist, so the problem in these instances cannot be that the checking query is fetching too much data.

Defining a Specification using Linear Regression

We now concern ourselves with understanding the precise relationship between the number of hashes to be checked, and the time taken by such a check. We begin by monitoring the Conditions Upload service with 4000 uploads with respect to the (very loose) specification

```
Forall(q = changes("hashes")).\
Check(lambda q : (
    q.next_call("find_new_hashes").duration() < q("hashes").length()
  )
)
```

Our intention is to determine the correlation between the quantities

```
q.next_call("find_new_hashes").duration()
```

which is the time taken in seconds by a specific call of `find_new_hashes` and

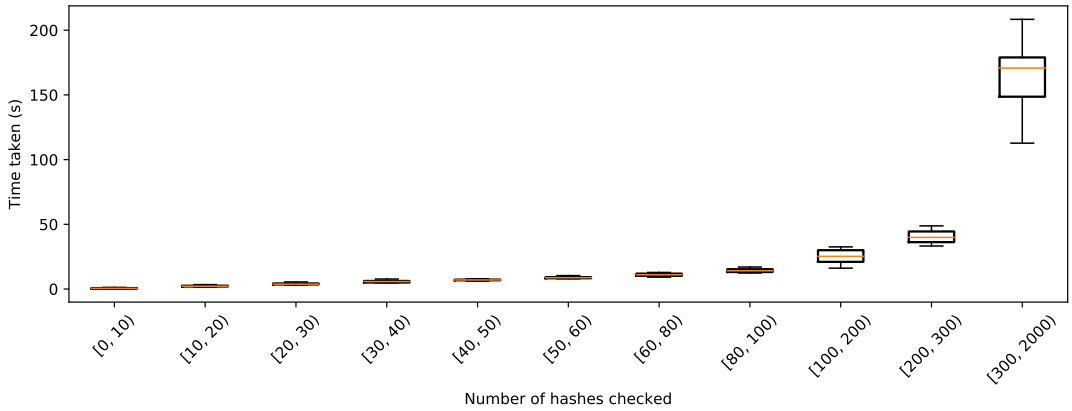
```
q("hashes").length()
```

which is the length of the list stored in the variable `hashes`. Given that we hope to be able to describe the relationship (without losing too much information) using a first order polynomial, we opt for a linear regression on the quantities recorded during the 4000 Conditions uploads. Figure 7.4 shows the quantities recorded along with a simple linear regression.

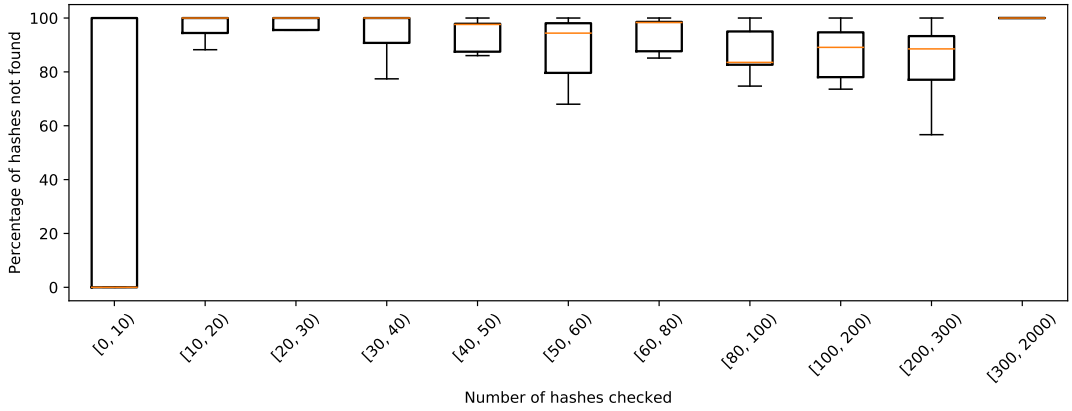
Using this approach, we derive the more precise specification

```
Forall(q = changes("hashes")).\
Check(lambda q : (
    q.next_call("find_new_hashes").duration() <\
      (q("hashes").length() * 0.157053 + 1.056350)
  )
)
```

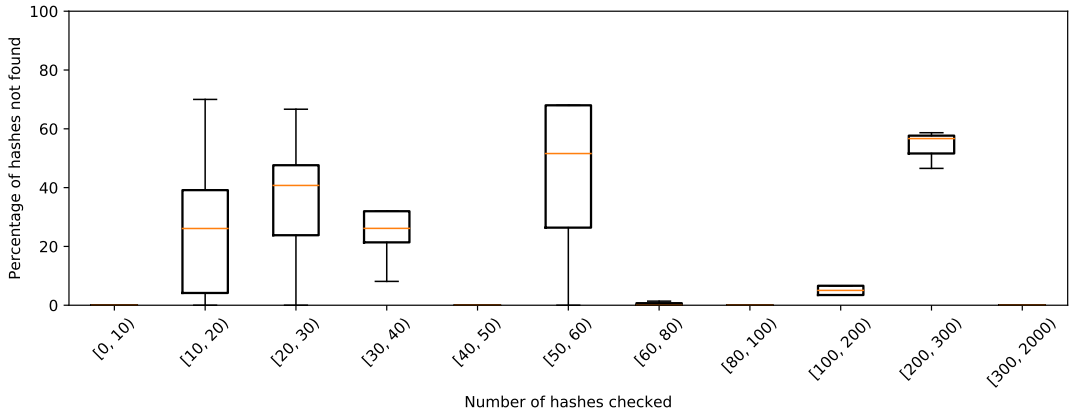
Using this refined specification, we rerun the Conditions Uploads to obtain new data. This time, we plot what we call the *severity* of observations, which is intuitively the distance between the



(a) The time taken by the hash checking optimisation with respect to the number of hashes that were checked.



(b) The average percentage of Payloads not found, with all data included regardless of verdict.



(c) The average percentage of Payloads not found, keeping only uploads for which fewer than 70% of proposed Payloads were not found.

Figure 7.3: Plots giving insight into the behaviour of the hash-checking mechanism with respect to the number of hashes being checked.

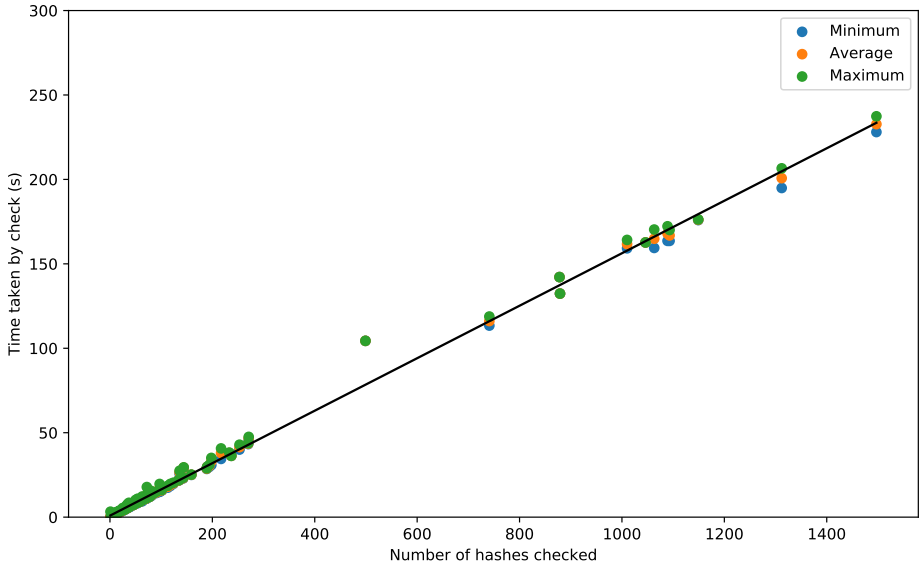


Figure 7.4: The time taken by the hash checking optimisation with respect to the number of hashes that were checked, with a fit added that was derived via linear regression.

measured value and the edge of the region of validity in the measurement's domain. Values that gave true verdicts are positive, and those that gave false verdicts are negative. Figures 7.5 and 7.6 show the severity of observations of time taken by hash checking with respect to the number of hashes checked. We see that, for smaller numbers of hashes (Figure 7.5), we do not have many violations and, in fact, it appears that the linear regression we performed works well. The absolute value of the severity of all observations is small. The situation is different for larger numbers of hashes (Figure 7.6). Here, we see that violations are slightly more frequent but also more severe. However, we also see that the uploads with the most hashes of the replay do not often cause violations. This agrees with our results so far: uploads with many Payloads are often sending new ones, so the performance drop observed for hash checking over large sets of hashes is less severe because it would not be affected by queries fetching too much information (ie, entire Payloads) from the database. Further, we should also expect some instability from the network on which the Conditions Upload service normally operates.

After applying the fix to the selection query used by the hash checking mechanism (minimising the amount of data being fetched), we generated Figure 7.7. We first remark that different behaviour was observed from the network at the time of this new experiment, so there is a greater range in times taken by the operation. We also remark that the severity in Figure 7.7 was computed based on the same fit as Figure 7.6, meaning the performance has not improved in the worst cases. Further, we see more signs of network instability in that some bins which previously generated very negative severities (therefore, severe failures) this time generated positive severities, and vice versa.

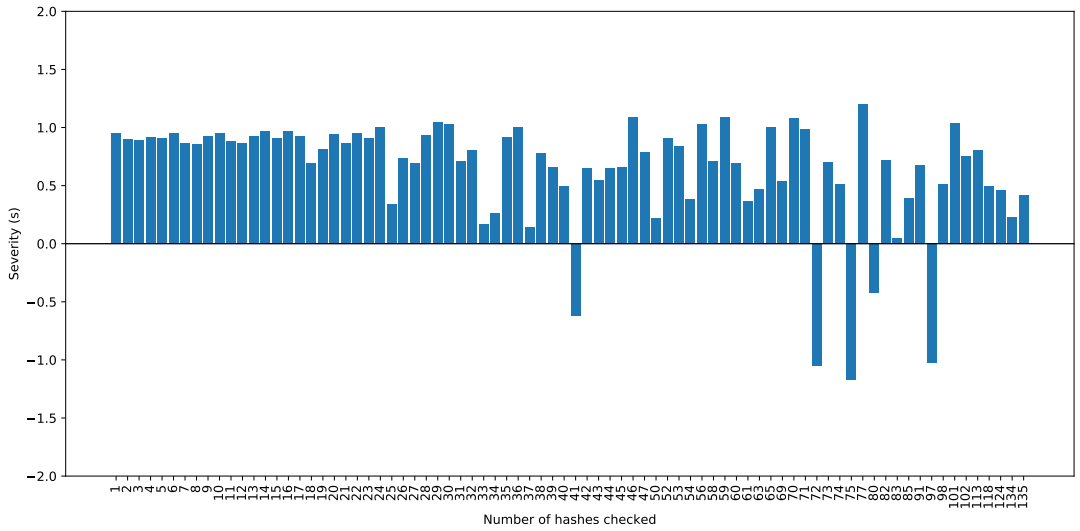


Figure 7.5: The severity with respect to the given constraint over the hash checking optimisation, restricted to smaller numbers of hashes.

Ultimately, the change made to the code before the data shown by Figure 7.7 was generated achieved the expected affect: the performance did not improve, because the performance problem was not caused by a query fetching data that was both very large and not needed. We now consider possible further development that will take place to address the remaining performance problem. While the change described here was made by this thesis' author, future changes will be made by the team now responsible for the Conditions Upload service.

We conclude from this final investigation into the hash checking mechanism that, since the problem was not overly-verbose queries, there are two remaining factors at the source code level. The first is the Object-Relational Mapping (ORM) used by the Conditions Upload service to execute its queries. This underlying library controls database transactions and generates queries for database operations expressed via the ORM. The second factor is our own code, which wraps the ORM in some helper functions. Since our use of the ORM already aims at avoiding potential bottlenecks (such as repeated reinitialisation of connections, rather than using a pool of open connections), we conclude this investigation with a recommendation that interaction with the central database for hash checking should be performed using pure SQL with no ORM layer. This modification could allow us to reduce roundtrips to the database, thus reducing the time taken by hash checking in the cases of thousands of hashes.

7.7.2 Payload Upload Stability

We would now like to understand the behaviour of the two connections involved in Payload uploads, since Payloads can be large (in the order of 100MBs) and have been a common source of problems during operational experience obtained so far. We remark that, here, we consider experiments running over a maximum of approximately 24 hours during periods of much reduced physics activity. Such experiments involved replays of the same 4000 uploads as our other

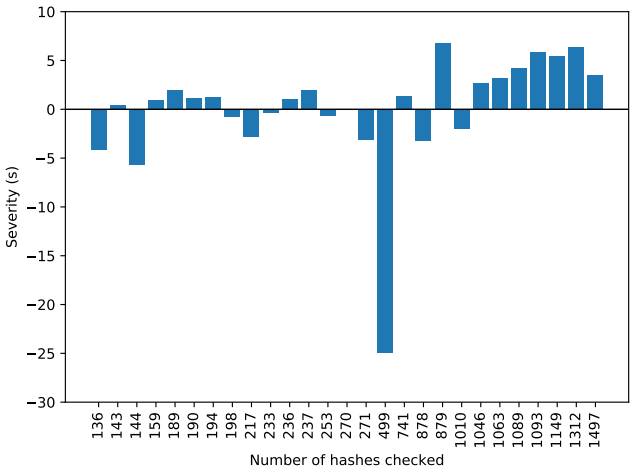


Figure 7.6: The severity with respect to the given constraint over the hash checking optimisation, restricted to larger numbers of hashes.

experiments, since these have been identified as containing sufficient diversity in data being sent. For example, some uploads send only one Payload, whereas others send approximately 1500. Further, some Payloads have size in the order of 10^3 bytes, whereas others have size in the order of 10^5 bytes.

While it is not appropriate to modify how Conditions are uploaded based on these short-term experiments, by taking measurements we can define some baseline for Payloads upload times. Using this, during the LHC runs for which the Conditions Upload service is scheduled to be deployed, we can use VyPR to monitor for deviations from this predetermined baseline.

Any information we get about the stability of the HTTP and database connections used is also important in determining for how long we should set the durations of upload sessions. The naive approach here would be to set a generous amount of time for the upload session, but if something were to go wrong leaving the session open, the next upload to the same Tag would have to wait a long time. Hence, to reduce the time waited in the case of an error (even if the code is thoroughly tested, network errors can leave sessions open), we must understand how long Payload uploads can take since these are the transfers that normally take the longest.

Our investigation involves using VyPR to record the times taken on the client and server sides of the Conditions Upload service and using call tree reconstruction to link these times up across machines. We use the specification

```
Forall(c = calls("self.send_blob")).\
Check(
  lambda c : c.duration() < c.input()("payload").length() * 0.000001
)
```

on both the client and server sides, where $(x \times 0.000001)$ is a function to be refined. With this specification, we express the requirement that the time taken to insert a Payload should be strictly less than some (very small) fraction of its size.

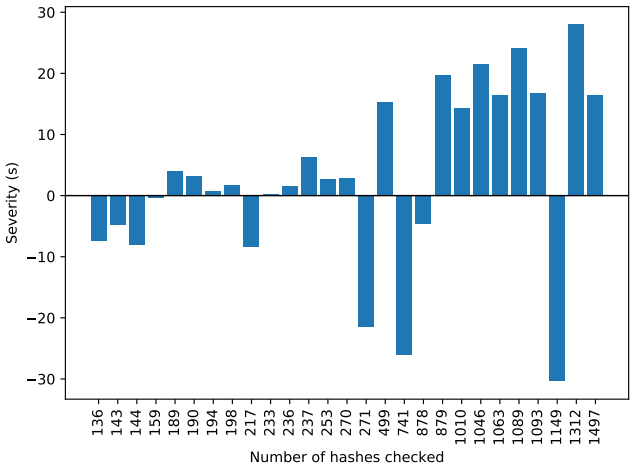


Figure 7.7: The severity with respect to the given constraint over the hash checking optimisation, restricted to larger numbers of hashes, after a fix was applied to a key query.

Using Call Trees

This analysis requires VyPR's call tree construction facilities, along with the ability to synchronise multiple instances of VyPR to a common clock, to allow call trees with function calls occurring on different machines to be constructed. Figure 7.8 shows the call tree derived after monitoring a single Conditions Upload consisting of sending two Payloads. We will investigate its exact structure. The root of a call tree derived from a system consisting of a client and a

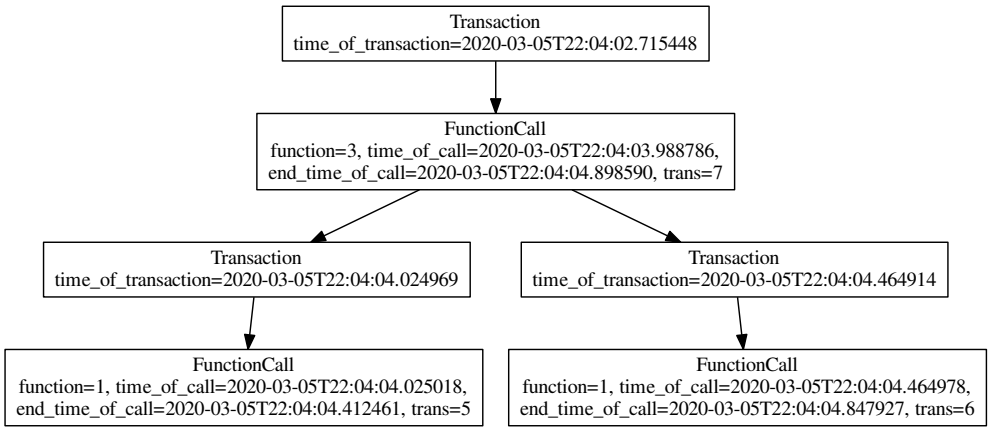


Figure 7.8: The call tree of a Conditions Upload during which the Payload upload functions on both the client and server-sides were monitored by VyPR.

server is always a *transaction* representing a single run of the client program. All child vertices of a transaction must be function calls, since transactions that are not the root must be triggered by some function call. There is no restriction on the type of child vertex of a function

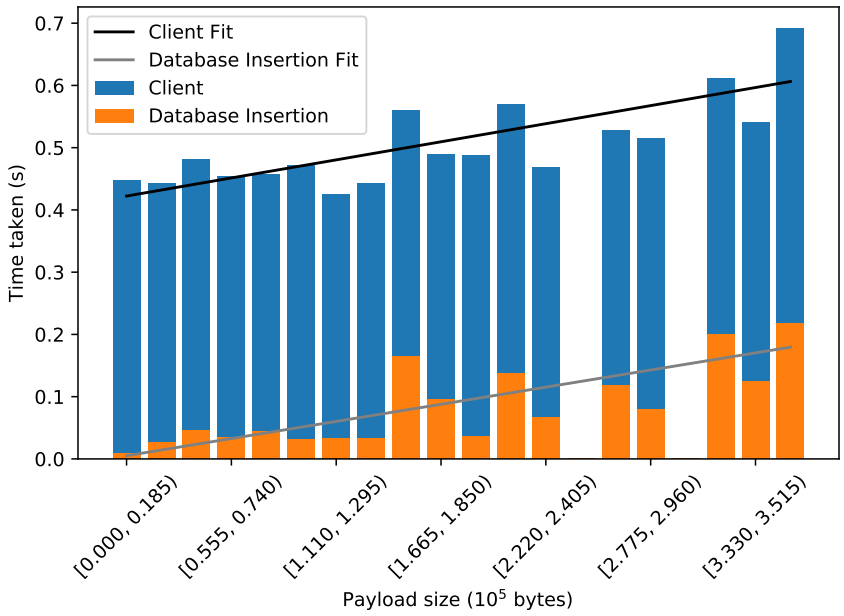


Figure 7.9: The time taken by Payload insertion from both the client and server-side perspectives.

call (since a function can call another function, or it can send an HTTP request to another machine, triggering another transaction).

In Figure 7.8, we see that a single function call can contain multiple transactions (or, rather, multiple transactions can take place during the same function call). This is the case if the function call involves multiple calls of other functions which communicate with another machine and induce transactions there.

Network and Database Behaviour in the same Plot

Running 4000 Conditions uploads, after post-processing using the analysis library, we obtained the plot in Figure 7.9, showing that the behaviour of database insertions is much more erratic than that of the HTTP connection needed to reach the upload server from the client. This behaviour exhibited by the database connection is consistent with the rest of the analyses presented in this chapter.

Post-processing involved binning the data generated by the 4000 uploads into ranges of Payload sizes in bytes and linear-regression on the times measured for HTTP and database insertion. The resulting bars representing HTTP transfers (measured on the client) have similar fluctuations to the bars representing database insertions, so we can say that HTTP transfers are more well-behaved than database insertions.

We now briefly look at refining our specification for future monitoring. Figure 7.9 shows two fits for the HTTP and database insertion data. The client fit can be used to determine the constraint to be placed on the client-side and the database insertion fit can be used for the server-side constraint.

Monitoring after Linear Regression

After performing the fit in the previous section, we arrive at a more refined specification with which to go forward into the commissioning stage of the Conditions Upload service. The server-side will initially be monitored with respect to the specification:

```
Forall(q = changes("blob")).\
Check(lambda q : (
    q.next_call("con.execute").duration() <\
      (q("blob").length() * 0.009178 + 0.005170)
  )
)
```

and the client-side with respect to the specification:

```
Forall(c = calls("self.send_blob")).\
Check(lambda c : (
    c.duration() < (c.input()("pl_blob").length() * 0.009684 + 0.422303)
  )
)
```

The constants chosen are based on the slopes and offsets of the lines determined by performing linear regression over the first dataset obtained. Additionally, the program variable names here have been changed to reflect the application to the different parts of the monitored service.

Further, these specifications make use of a number of features provided by VyPR. We see, as before, the ability to compare multiple quantities measured at runtime. We also see the ability to apply functions to those quantities. Further (dynamic) refinement of these specifications falls under the umbrella of *Specification Tuning* and will be explored further during the main commissioning period of the new Conditions Upload service, which will run into 2022.

7.7.3 Metadata Validation

IOVs are inserted during a process that depends on whether the proposed destination Tag for the Conditions exists. In particular, if the proposed destination Tag does not exist, there are no constraints on the IOVs to be inserted. However, if the proposed destination Tag does exist, the IOVs must be checked and this process depends on characteristics of the existing Tag.

Addition of IOVs to certain Tags requires communication with other machines in the surrounding network for validation, whereas other Tags may not require any communication. In the case that validation must take place, the checks performed are usually with respect to the times to which the IOVs correspond. For example, some types of Tags do not allow new IOVs to be inserted into intervals covered by existing ones.

With this in mind, we would like to analyse the time taken to get from the first query that checks for existence of the proposed Tag to the final commit that takes place to finally write the IOVs to the database. We assume that there are two branches that can be taken in the monitored source code between our two statements of interest: *branch 1* is the branch taken when the Tag already exists, and *branch 2* is the branch taken otherwise. Our specification is as such:

```
Forall(q = changes("tag")).\
```

```
Forall(c = calls("commit", after="q")).\
Check(lambda q, c : timeBetween(q, c.result()) < 1.0)
```

This expresses the requirement that, between every pair consisting of a change of the variable `tag` and a call of the function `commit`, the time taken to reach the end of the call of `commit` from the change of `tag` should be strictly less than 1 second.

Once we have monitored the Conditions Upload service with this specification, we can use one of the VyPR analysis scripts to perform comparison of the paths between changes to `tag` and calls to `commit`. To understand the analysis that will be performed, we must think about the results of instrumentation and monitoring.

Understanding the Analysis

Since there are two quantifiers, we expect each static binding generated by instrumentation to be a map with domain $\{q, c\}$ (see Chapter 4 for the details of this). Further, using knowledge of the code being monitored, we expect there to be two static bindings generated, though we will only concern ourselves with the information we get from one. There is also no loop, so monitoring will derive at most two bindings from the dynamic run generated by the code; one for each static binding determined by instrumentation.

We can also make the observation that, since there are no `next` operators used in the quantifier-free part of the specification, the only parts of the program we instrument will correspond to the images of the variables q and c under static bindings. Hence, the grouping that we must perform to ensure that the paths between the concrete states `q` and `c.result()` generated at runtime start and end at the same statements in code is simply a grouping with respect to static bindings.

Once we have the paths leading to `q` and `c.result()` for each static binding, we further group by verdict and subtract the paths leading to `q` from those leading to `c.result()`. We then use the existing machinery to compute the intersection (see Chapter 5). The weight of our analysis will be in how we deal with the resulting correspondence between verdicts and parametric paths.

In particular, in this case the intersection we perform will yield a path with at most one parameter. If all paths for a single verdict are identical, we obtain no parameters. If there is at least one path that deviates from the rest, we will obtain at least one parameter. Given that the code that we are analysing contains a single conditional (with one branch containing another nested conditional), the most likely situation on the first pass of our analysis is that we obtain a parametric path for each verdict with a single parameter. This parameter will be a result of disagreement at the conditional.

Generating Plots with respect to Path Parameters

We will now demonstrate how our explanation approach described in Chapter 5 can be applied in order to analyse performance with respect to program paths traversed at runtime. Our goal is to determine whether a single path is frequently to blame for poor performance. We will take the following steps:

1. Get all pairs of measurements (for `q` and `c.result()`) recorded in the cases that our specification was violated.
2. For each pair, reconstruct the path taken by the program run to reach `c.result()` from `q` (since both `q` and `c.result()` refer to the state of the program immediately after some statement was executed).
3. Determine where the resulting set of paths disagree (in this case, the single if-statement at which two branches can be taken). This point of disagreement will be encoded as a *path parameter*.
4. Determine the set of distinct branches taken. Each branch will be encoded as a *value* of the path parameter.
5. Group the measurements by the branch that was taken to generate them.
6. Construct two plots for each branch. This will allow us to see whether there is a clearly problematic branch with respect to measurements taken.

Step 1. Assuming that the analysis library's package is available via the variable `va`, we begin by selecting the function/property pair.

```
metadata_function = va.list_functions()[0]
```

With this function object obtained, we construct the symbolic control-flow graph of the function we are analysing and then derive its context-free grammar using

```
scfg = metadata_function.get_scfg()
grammar = scfg.derive_grammar()
```

For each call of the function, we now determine the failing verdicts (for which we obtained an object) that were generated by the part of the code we care about (by supplying a binding).

```
binding = metadata_function.get_bindings()[0]
calls = function.get_calls()
for call in calls:
  verdict = call.get_verdicts(verdict=0, binding=binding)[0]
```

Then, inside the loop, we get the set of two observations that were recorded by VyPR for each verdict.

```
  observations = verdict.get_observations()
```

Step 2. We determine the set of all paths leading from `q` to `c.result()` for each pair of measurements. With VyPR's analysis library, this is straightforward. We opt to use some of the library's lower level functions to demonstrate in more detail how the internal machinery is being used.

For each call, and each verdict generated by that call, we reconstruct the paths taken to reach each observation required to reach a truth value for that verdict using

```
  lhs_path = observations[0].reconstruct_reaching_path(scfg)
  rhs_path = observations[1].reconstruct_reaching_path(scfg)
```

As before, the symbolic control-flow graph is parameterised during any path comparison operations. The next major operation we perform is the construction of the parse tree of the path difference, using

```
path_difference = rhs_path[len(lhs_path):]
parse_tree = ParseTree(path_difference, grammar,
                       path_difference[0]._source_state)
```

Step 3. Once we have a list of such parse trees for each path, we assume that such a list is stored in the variable `parse_trees` and obtain the intersection using

```
intersection = parse_trees[0].intersect(parse_trees[1:])
```

The intersection parse tree we obtain is shown in Figure 7.10. The single path parameter here is highlighted in black: this leaf holds a vertex from a symbolic control-flow graph, rather than an edge. To determine the path to this subtree that we will use to determine values given to the path parameter, we use

```
path_parameters = []
intersection.get_parameter_paths(intersection._root_vertex, [], path_parameters)
```

The variable `path_parameters` will hold a list of lists of edges. Since a central structure in our explanation approach is the *parse tree* (see Chapter 5), these lists of edges act as paths through the relevant parse trees. These paths through parse trees are precisely those described in Chapter 5.

Step 4. For any parse tree used in the intersection, we obtain the subpath that it gives to the path parameter by

```
subtree = parse_tree.get_parameter_subtree(path_parameters[0])
subpath = subtree.read_leaves()
```

which involves first determining the subtree found by following the parameter's path through the parse tree, and then reading the leaves of this subtree. Further, notice that the subpath generated by reading the leaves of the subtree can be generated by a rule in the grammar. This rule is precisely that which is associated with the symbolic state held at the root of the relevant subtree.

We omit steps 5 and 6 because they make no special use of the VyPR analysis machinery. The resulting plot is given in Figure 7.11, where we have applied the procedure described above to all measurements that generated a failing verdict. The experiment that generated this data involved a replay of 4000 uploads.

A Note on Usability

We comment now on the amount of effort that was required to perform the analysis that was presented above. While the analysis library takes a significant step towards making analysis more straightforward (for example, performing the analysis above by querying the verdict server's database manually and then performing the path reconstruction and intersection would

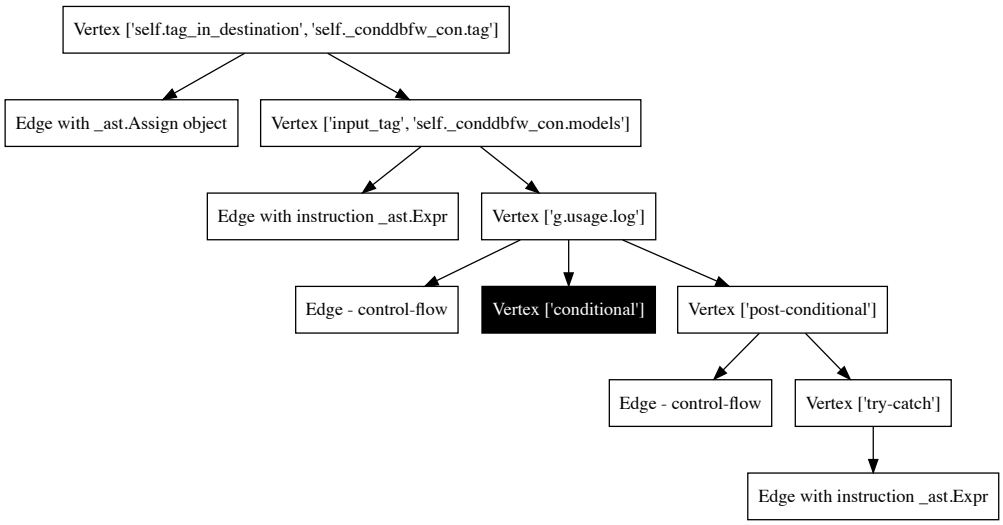


Figure 7.10: The parse tree intersection derived from the Conditions Upload service code that allowed us to separate verdict data by path parameter values.

take far more effort), the effort required is still too much for a software engineer who wants to get fast insights into their code's performance with minimal effort.

To help with the situation, we introduced the web-based analysis environment (see Section 6.8) which allows common analyses to be performed with a few clicks (and little knowledge of VyPR's internals).

Interpreting the Plot

In Figure 7.11, the plot at the top shows more violations that are severe than the plot at the bottom. The top plot shows all timing measurements taken along the branch that found an existing Tag and so had to perform validation according to some criteria (defined by the type of upload). While it is expected that this branch will be less performant, the severity of a large number of the violations is concerning and indicates that we should be aware of this part of the upload process as the Conditions Upload service is being moved into production.

The immediate suspicion is, given that the paths traversed during measurements included the final database commit, and this is likely to be slower for larger amounts of data being committed, the branch taken by measurements found in the top plot often involved larger amounts of data.

We make a final observation that there was no communication with external machines to validate proposed metadata in these experiments. Rather, the results of such communication was recorded along with the original uploads and simulated. So we can be sure that communication with other machines on the network (apart from database servers) was not a factor.

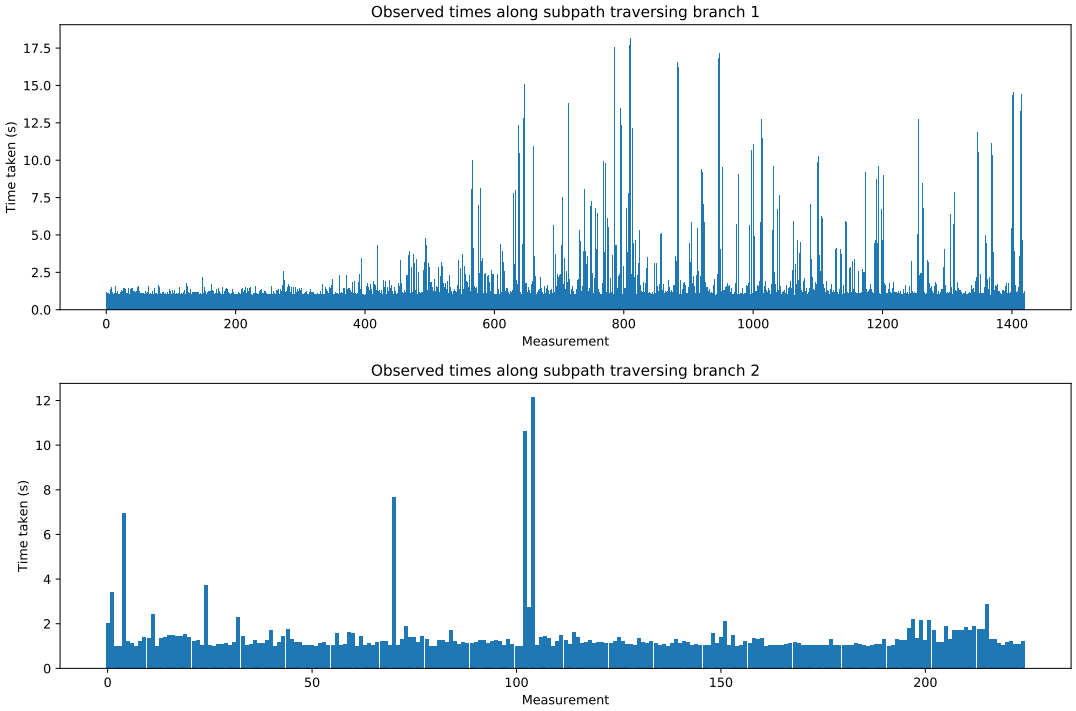


Figure 7.11: Plots showing, in the case of failure, how much time was taken to reach `c.result()` from `q`. Each plot shows measurements taken along a distinct path.

Extra Monitoring along the Path of Interest

In order to investigate the results shown in Figure 7.11, we repeated monitoring with instrumentation of critical function calls along each path. Notice that simply profiling the functions themselves, with no context taken from call sites, would make the analysis we now present very difficult. Instead, using VyPR, we simply place an arbitrary specification over the functions that we care about and exploit the fact that VyPR stores its measurements. The necessary post-processing is then a simple extension of what we have already; for each path measurement, we include the corresponding measurements of durations of two other critical functions.

To obtain new measurements, we first adjust our initial specification in light of the fact that many measurements are included in Figure 7.11 among which there seems to be little fluctuation. The new specification is

```
Forall(q = changes("tag")).\
Forall(c = calls("commit", after="q")).\
Check(lambda q, c :
  timeBetween(q, c.result()) < 1.2
)
```

with other arbitrary constraints included to obtain measurements of other critical function calls. The result of monitoring is Figure 7.12. Our first observation is that the looser constraint placed by our specification fits the behaviour of the program better. We see this by realising that there are fewer points included in the plots which contain only measurements that caused

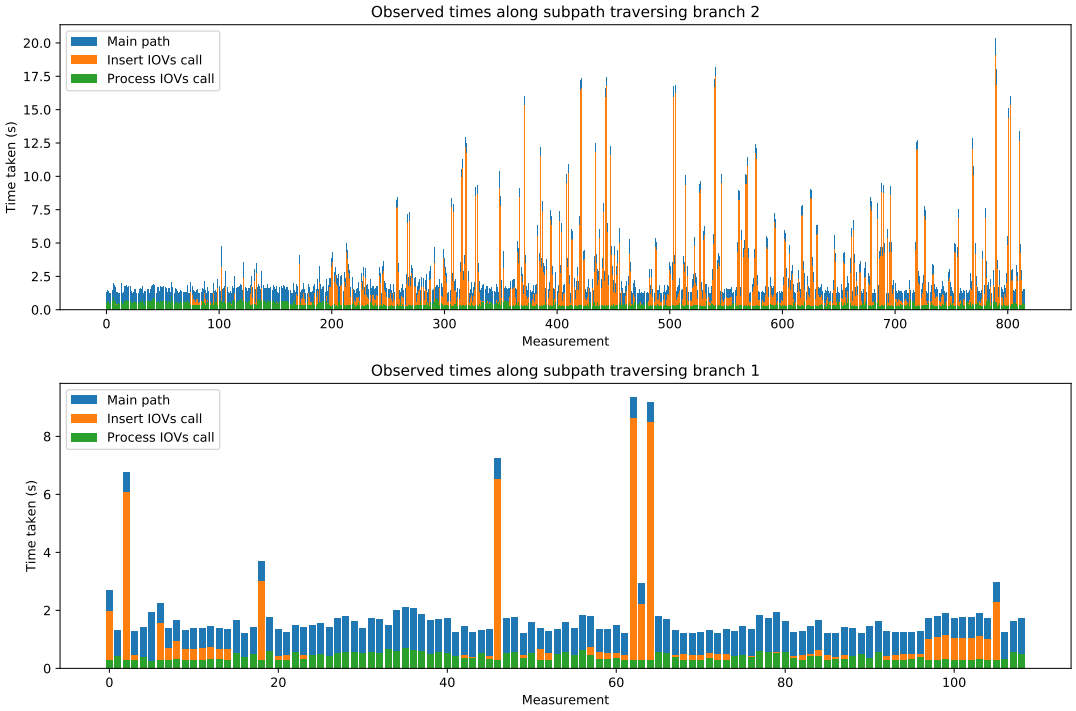


Figure 7.12: Plots showing, in the case of failure, how much time was taken to reach `c.result()` from `q`, with contributions from critical functions called along the way.

violations. Our second, key observation is that calls to the function responsible for *inserting IOVs* often show similar behaviour to the measurements along the path that we care about, while the other measurements remain approximately constant. Given that the IOV insertion process must iterate through every IOV, it is susceptible to large increases in time taken but, again, we highlight that the times shown in Figures 7.11 and 7.12 are higher than expected.

7.7.4 Monitoring Multiple Access Control

The Conditions Upload service employs a mechanism called a *Tag Lock* to ensure mutually exclusive access to each user to a single Tag. The reason for this is that, if multiple users were to upload IOVs to the same Tag at the same time, the resulting final state of the Conditions database could be different depending on the order in which the IOVs were sent. The Tag Lock mechanism ensures that a predictable outcome is achieved; there is no interweaving of IOV insertions from different users, unless they are from uploads to separate Tags.

At the beginning of each upload, there is a specific request to the Condition Uploader that asks for an upload session. We would like to gain insight into the performance of this step. The specifications that we use are

```
Forall(
  t = calls("Usage")
).Check(
  lambda t : (
    t.duration()._in([0, 0.1])
```

```
  )
)

Forall(
  c = calls("Usage")
).Check(
  lambda c : (
    timeBetween(
      c.input(),
      c.next_call("g.usage.new_upload_session").result()
    )._in([0, 1])
  )
)
```

Instead of using the verdicts obtained by VyPR's monitoring algorithm at runtime, we imposed a filter in post-processing because we were interested in the difference between the `timeBetween` operator's result and the duration of the call of `Usage`. This had to be done in post-processing because the current version of VyPR does not support the comparison of the result of the `timeBetween` operator with another quantity measurable at runtime. We remark that extending VyPR to be able to perform such a comparison is possible (and is planned if more cases are observed where it would be useful), but in this case performing the check in offline analysis was as effective.

The resulting plot is given in Figure 7.13. The constraint that we imposed in post-processing was the disjunction of:

- The difference between the `timeBetween` operator's result and `t.duration()` should be greater than 0.12 seconds.
- The result of the `timeBetween` operator should be greater than 1 second.

This allows us to see whether the instantiation of `Usage` was to blame for any increases in the value observed for the `timeBetween` operator. From Figure 7.13, we can see that, in most cases, increases in time observed for the `timeBetween` operator did not match increases in the time taken by instantiation of the `Usage` class. From this, we conclude that the instantiation of `Usage` is not problematic along the path that we studied. We therefore conclude that the actual upload session creation which occurs later in the path than the instantiation of the `Usage` class is more likely to blame for any increases in time taken.

7.8 Performance of VyPR

We now discuss the performance of the VyPR project as a whole when used to analyse the Conditions Upload service. We will discuss both the offline analysis library, and the online monitoring. We omit a discussion of instrumentation because this is done statically and has never been performed in any situation with a strict time constraint. Further, instrumentation took a matter of seconds in all of the cases that we ran it.

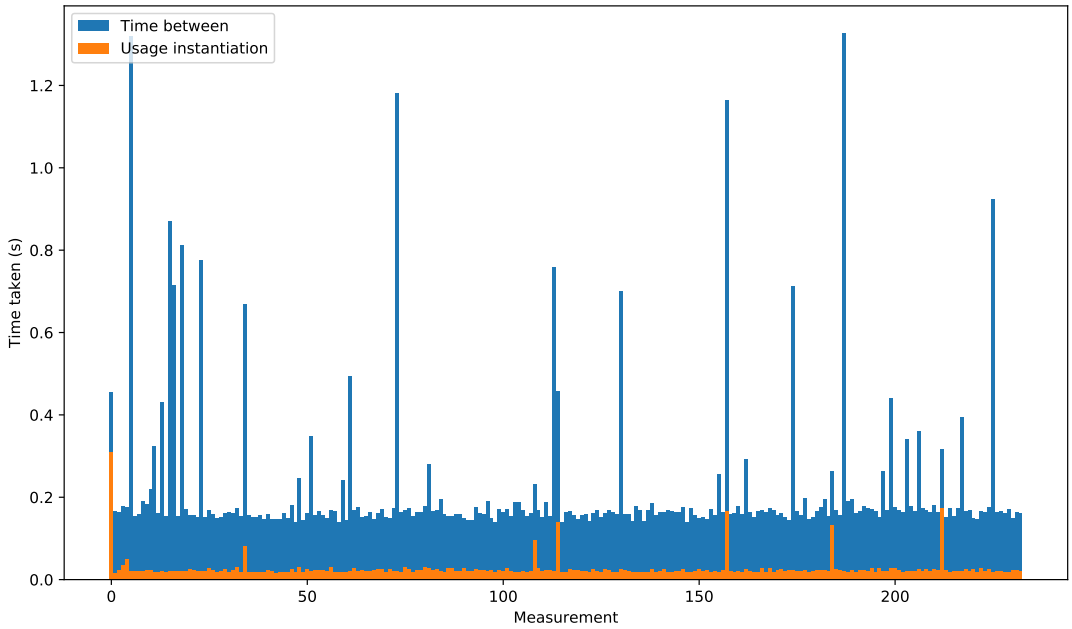


Figure 7.13: The time taken between two points in the mechanism that creates new upload sessions, with measurements added from the instantiation of the `Usage` class.

7.8.1 Online Monitoring

The precise overhead induced by VyPR is not straightforward to determine when it is applied to a system that often communicates with other machines over a network and deals with such a large range of inputs. In general, the overhead is observed to be small because of the instrumentation and monitoring algorithms employed. In obtaining a precise number, given a system like the Conditions Upload Service, we observe two difficulties:

- The results of monitoring depend on the surrounding network whose behaviour is not constant. Early attempts at overhead measurement showed that simply replaying a large set of uploads and comparing the times taken with and without VyPR is not a good approach and can give very different numbers from one run to the next.
- The upload process itself (and thus the access to the surrounding network) depends largely on the input data.

  For example, certain types of Conditions require additional queries to the target database, and can even require communication with an external machine at CERN's Tier0 to determine a quantity that we call the *first conditions safe run* that is used in IOV validation (though this check was not performed during our experiments). Hence, the results of monitoring IOV validation depend on the category of Conditions data.

In the following sections, we present analysis of VyPR under various scenarios. Our analyses show that VyPR in general induces low overhead on Conditions data uploads, typically below 5%. In a setting where a complete upload can take up to 300 seconds, 5% of this time being

15 seconds, this overhead is not a lot. Our main source of evidence for this is that engineers across 3 teams at CERN indicated that 5% is an acceptable amount of overhead, and this can also be seen in the report from the RV 2014 competition [BFB+17].

Further, we highlight that much of the time taken by uploads is actually taken up by network operations, during which VyPR's monitoring mechanism is given control by Python's thread management system. Hence, 5% here would be an upper-bound and some of the 15 seconds would be spent while information was being sent over the network connections involved in a Conditions upload.

Finally, we highlight that experience with VyPR on case studies not described in this thesis has shown us that engineers are happy with 5% overhead or below.

Overhead of Asynchronous Monitoring and a Proposed Solution

When monitoring is asynchronous, the problematic time overhead does not cause problems for the program being monitored, rather for the delay with which monitoring results are available.

During extensive experiments, we have observed that the (currently primitive) architecture of the verdict server can sometimes act as a bottleneck. The effects of this bottleneck go sufficiently far back to be visible in the size of the consumption queue used by VyPR to enable communication of measurements by code in the monitored program with the monitoring algorithm. In particular, we can observe the following chain of events:

1. Since our current verdict server implementation uses an SQLite database (a file on disk), if the event rate from the monitored program is high enough and the IO performed on the database slows down, this latency affects the response from the verdict server in the monitoring thread.

2. The latency experienced by the HTTP request sent by the monitoring thread slows down processing of the queue populated by the instrumentation code in the monitored program.

3. If the latency experienced by HTTP outweighs the event rate, the consumption queue must fill up faster than the monitoring algorithm can process the events. This phenomenon in itself can only happen in the asynchronous setting. If the monitoring algorithm were synchronous, the monitored program would block and the induced overhead would be unacceptably high.

4. Given that the monitoring algorithm is asynchronous, the result is a backlog of events to process by the (still living) monitoring thread once the execution of the main program has finished.

A straightforward solution to this could be to make the HTTP requests on the client-side asynchronous (ie, start a new thread specifically for handling HTTP requests). However, this new thread would still experience latency and the running time of the process of the monitored program would be affected in the same way. Our planned solution is therefore to decouple the end-points on the server from the database operations so that the database operations need not be completed before a response is sent to the request's origin. With this approach, we lose

the ability to report problems with insertion directly to the monitored program, but we also eliminate the latency induced by the database operations.

Monitoring without Explanation

Monitoring without additional instrumentation for explanation has been shown by operational experience to be quite efficient. By running the same Conditions upload 10 times (this guarantees that the actual upload process is the same, since the process varies from upload to upload), we obtained an average time overhead of approximately 2.8%, with lower overheads being towards 1% and higher ones being towards 5%. The source of uncertainty in this measurement is network instability. This involved two properties, each over a separate function in the Conditions Upload service code.

Monitoring with Explanation

As we have seen, explanation necessitates additional instrumentation to record paths taken by the relevant functions during program runs. Here, we will use the same approach as above (rerunning the same upload), but we will pick one that generates a large amount of output from path recording instruments, hence placing more strain on the monitoring mechanism provided by VyPR.

We run an upload containing 39 Payloads 10 times with path recording turned on to be able to determine the number of Payloads being checked by hash checking (as we did in Section 7.7.1). We recorded an average time overhead of approximately 0.5% and make the important observation that, despite the slightly higher load placed on the monitoring thread by hash checking, the additional IO required to insert the higher number of Payloads allowed VyPR's monitoring thread to perform its logic asynchronously, leading to almost non-existent overhead. Further, when considering how much instability of the surrounding network affects these overhead measurements, we observe that 0.5% is insignificant compared to the induced error. Ultimately, we instead opt for a description of the overhead as *negligible*, unless a situation like that described in the next section is observed.

How to Increase Overhead

We finish with a discussion of the cases in which the overhead induced by VyPR is known to increase. These cases are defined by the behaviour of the Global Interpreter Lock (GIL) used by Python as a mutual exclusion-based mechanism for avoiding concurrency problems such as data races. The GIL is held by one thread at a time, meaning that no other threads can execute. However, if the thread holding the GIL performs IO (such as sending an HTTP request), the GIL is released and other threads can work. As discussed in Section 6.4.4, if a program performs frequent computations without any IO, meaning the Global Interpreter Lock rarely gives control to any other threads, the overhead is likely to be high, since VyPR would not get a chance to progress. In cases like this, the amount of code executed by VyPR may even be more than that executed by the monitored program (if an engineer is monitoring a small, short-running program).

This limitation is possible to overcome, either by using multiprocessing (hence, running VyPR outside the control of the GIL) or by porting VyPR to C/C++ and bypassing the GIL using established techniques [pyta]. We have not addressed this yet because our use cases have been web services, hence involving heavy IO operations.

7.8.2 Offline Analysis

We will now examine the efficiency of multiple offline analyses approaches used throughout this chapter. The first analysis script (Section 7.8.2) involves applying our path comparison approach (see Chapter 5) to determine problematic paths with respect to results of monitoring for a specification. The second script (Section 7.8.2) involves constructing call trees based on specifications written over multiple functions.

We highlight that, so far, no special treatment is given by the verdict server to *bulk analyses* (analyses requiring information from many different rows in a table), hence the necessary round-trips added by many HTTP requests add to the time taken by analyses, though not restrictively so.

Before presenting the analyses, we remark that use of VyPR's analysis facilities is always performed offline and therefore does not contribute to the overhead induced by VyPR during monitoring. As such, the results shown here are to give insight into how VyPR would currently perform when integrated into the software development process.

Path Comparison

We first examine the time taken by the script that we use to compare paths in the case of metadata checking (see Section 7.7.3), because path comparison is a typically costly operation; it involves iteration over a symbolic control-flow graph, along with parse tree construction and intersection. In fact, we expect parse tree intersection to be the most expensive operation since this involves iteratively refining a set of paths through a parse tree.

Figure 7.14 presents a plot of the time taken by the metadata path comparison analysis from 100 to 3000 calls of the metadata check function, incrementing by 100 calls at a time. We see immediately that the time taken scales approximately linearly. We observe some key facts that affect this linear scaling:

- Derivation of the context-free grammar (from the relevant symbolic control-flow graph, see Chapter 5) over which parse-tree construction is performed, is amortised and need not be performed for every function call.

  We observe that path comparison does not currently allow analyses of paths through different versions of the source code the monitored program, so the overall process may change.

- The construction of a parse tree is quite efficient since the grammars derived from symbolic control-flow graphs are not ambiguous, so the generating rules can be selected from the grammar using a one-step lookahead. Further, for each call the construction is independent of other calls.

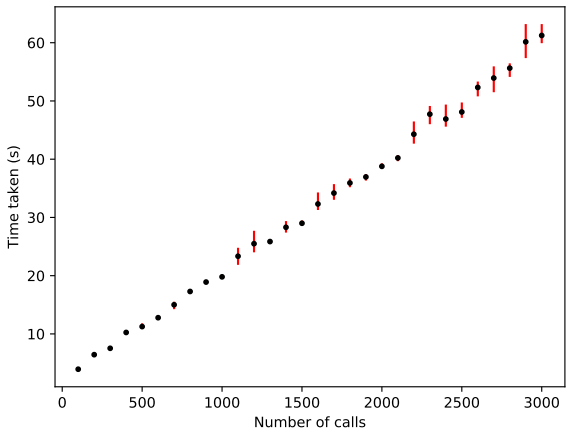


Figure 7.14: The time taken by our metadata check analysis script.

- Intersection of parse trees in order to determine regions of disagreement in paths (see Chapter 5) is usually quite efficient because, often, the entirety of every parse tree involved need not be considered. In particular, if the parse tree that contains the fewest vertices is encountered early on in the intersection process, intersection is especially efficient because there aren't many common paths to check for the other parse trees.

  While we have not yet needed to optimise this process, one could 1) order the parse trees by size or 2) move the smallest one to the front to increase the chances of the paths processed being refined very heavily, early on.

Finally, we observe that path intersection will often not take place over 3000 calls at the same time, rather intermediate results can be stored over time. This is possible due to the commutativity of intersection.

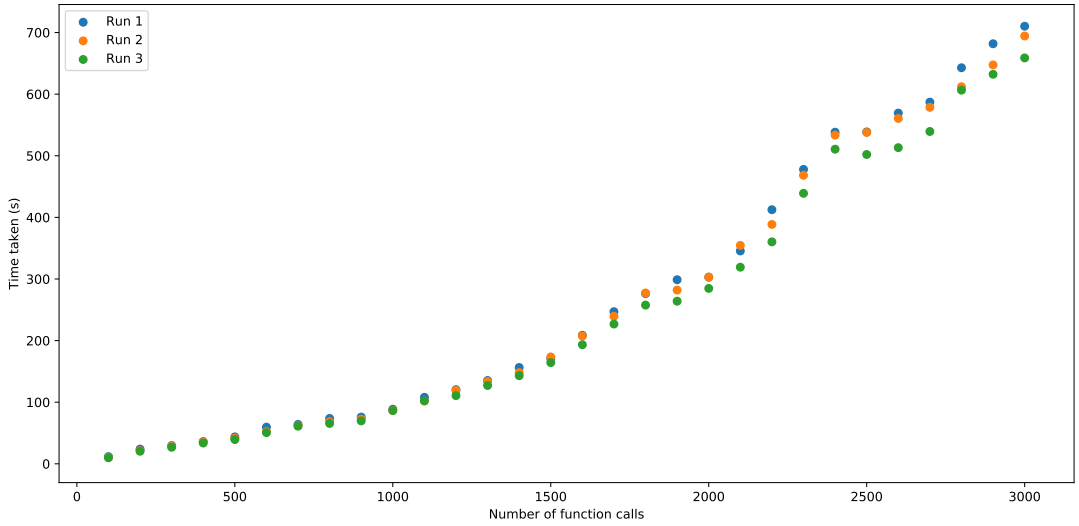
Call Tree Construction

We now examine the time taken by the script that we use to determine how often Payload uploads take much longer than expected. We expect the time taken by this script, without any optimisations applied to the way in which transactions and function calls are stored, to scale worse than linearly with the number of calls to process.

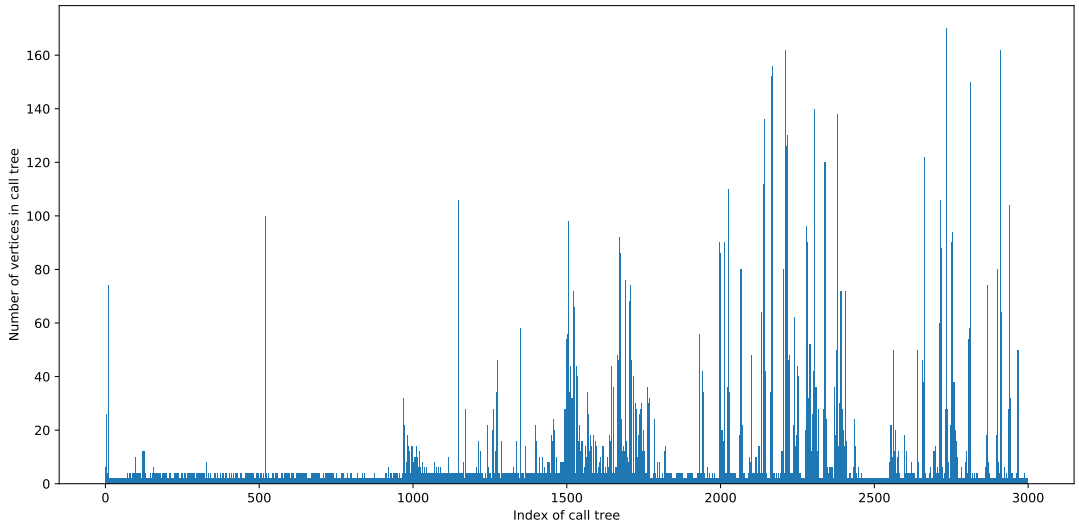
However, we must make the observation that the number of calls is not the only factor in the time taken by call tree reconstruction across multiple transactions. The other fact is the size of the call trees (ie, the number of vertices) being constructed. In the case of the Conditions Upload service the size can vary greatly between uploads, with some call trees containing just a couple of vertices, but others containing in the order of 100.

Figure 7.15a shows a plot obtained by running the Payload upload analysis script with 100 to 3000 Conditions uploads, each time increasing the load by 100 calls. To understand the shape of the curve given, we take into account the plot shown by Figure 7.15b.

Figure 7.15a was generated by running increasingly large subsets of one global set, such



(a) The time taken by our network latency analysis script.



(b) The size of the constructed call tree with respect to the transaction being used.

Figure 7.15: Plots used to analyse the scaling performance of our network latency analysis script for Payload blob uploads.

that each set used was a subset of each of the larger sets. We now look at Figure 7.15a at the regions between 1500 and 2000, 2000 and 2500, and then 2500 to 3000. We consider the same regions in Figure 7.15b. We make the following remarks:

- For the region 1500 to 2000 in Figure 7.15a, there is a dramatic increase in the size of the call trees constructed, reflected in Figure 7.15b. Towards the end of this region in Figure 7.15a, the call trees become small again and the slope of the curve drawn out in 7.15a therefore becomes shallower.
- We observe the same effect between 2000 and 2500, where we see a large increase in the

size of the call trees and, as a result, a large increase in the time taken by the script shown in Figure 7.15a.

- The result is similar between 2500 and 3000.

Ultimately, Figure 7.15b acts as a derivative for the curve drawn out in 7.15a.

7.9 Conclusions

Over the course of this chapter, we have demonstrated extensive investigations of the CMS Conditions Upload service that have led to significantly improved understanding of its performance. We have also seen the capabilities of the VyPR framework, developed using the theory introduced throughout this thesis.

7.9.1 Insights into the Conditions Upload Service

During our investigations of the Conditions Upload service, we have clarified our understanding of several parts, including:

- **Hash checking**, an important optimisation that we now suspect can be made optional to improve the process applied to Conditions containing many Payloads.
- **Metadata validation**, the part of the upload process that involves ensuring that data proposed alongside Payloads, such as Tags and IOVs, are correct. We are now aware that uploads to existing Tags are more prone to performance drops and will seek to improve this situation as the Conditions Upload service is commissioned.
- **Payload upload stability**, a part of the upload process that has historically caused problems when unstable networks are present. We have taken the first steps in determining where instabilities lie between the two major network operations performed: client-server communication, and database insertion performed on the server.
- **Multiple access control**, a part of the upload process that is crucial to predictable outcomes of uploads. Our investigation here determined that the main object instantiation corresponding to access control is not a performance bottleneck, rather the problem is likely to lie with the database queries used to open new upload sessions.

7.9.2 Use of VyPR

This chapter has demonstrated most of the capabilities of the VyPR framework. We have seen how CFTL is used in performance analysis of real-world software and how the instrumentation and monitoring infrastructures provided by VyPR allow sophisticated offline analyses based on CFTL specifications.

Further, we have demonstrated how VyPR can easily be used to perform a wide range of analyses, including measurements of durations of function calls; measurements of variable

values at critical points in code; and analysis of paths to facilitate explanation of observed performance.

Ultimately, we have shown 1) the effectiveness of CFTL as a low-level specification language and 2) the importance of powerful analysis tools for exploration of monitoring results.

Chapter 8

Conclusion

This thesis has introduced a new formal approach to the behaviour analysis of programs in the frame of Runtime Verification. This approach involves:

1. Writing *specifications* of program behaviour in our new language, Control-Flow Temporal Logic.
2. Using the VyPR ecosystem to instrument and monitor for agreement of a program with the specifications.
3. Using the VyPR ecosystem further to perform offline analysis of the results in order to determine root causes of disagreement with the specification.

Our emphasis has been on the analysis of Python programs, and then on making such analysis straightforward for engineers. This has involved addressing problems with multiple parts of the existing Online Runtime Verification process.

8.1 Usability in Runtime Verification

Runtime Verification exists because of a need for a *light-weight formal method* that can be used easily by engineers in the software development process. Where most previous work in the area has focussed on providing expressive specification languages (often at the cost of ease-of-use for engineers), we have acknowledged the importance of the analysis facilities provided by any tool released in the area of Runtime Verification.

8.1.1 Specification

Prior to this work, many specification formalisms existed in the form of temporal logics, rule systems, streams and automata. However, most exhibited one or both of the two problems that we define in [DR19b]: *separation* and *expressiveness*.

We say that those operating at a high-level of abstraction with respect to the program being monitored have the *separation* problem, meaning that effort is required to:

- *Define* how properties captured in those formalisms relate to programs. This means that, in order to express a property over the behaviour of a program, a definition must be given for each part of the specification (in terms of events that happen at runtime) in order for monitoring to take place.
- *Interpret* how properties captured in those formalisms relate to programs. This means that, given a specification and a program, the property expressed by the specification sometimes cannot be immediately understood because the events occurring at runtime that the specification actually describes cannot be known without more information (usually an instrumentation mapping).

These problems can make integration of a specification approach into a software development process tricky.

Further, we say that languages employing syntax such as complex temporal operators can suffer from the *expressiveness* problem. Formalisms with this problem can allow software engineers to express very complex properties but can end up making specification of much more common properties overly difficult.

To address these two problems, we introduced a new specification language, Control-Flow Temporal Logic, which is 1) low-level, having the novel characteristic that it also acts as an instrumentation description language; and 2) easy to use to express simpler properties. We acknowledge that the second point results in a lack of expressiveness, for example CFTL loses some expressiveness because of its lack of complex modal operators such as *eventually* and *until*, but this did not prevent us from expressing the properties that we required during the evaluation described in Chapter 7. We further highlight the importance that we placed on our analysis tools: while CFTL does not provide operators that lead to higher expressiveness, our analysis tools enable much more complex investigations once monitoring results have been obtained.

8.1.2 Instrumentation and Monitoring

While typical work in Runtime Verification has introduced automatic monitoring procedures, instrumentation is rarely automatic and often requires manual effort to relate a specification to the program that would be monitored.

Introduction of Control-Flow Temporal Logic naturally led to new instrumentation (which itself is an under-developed part of Runtime Verification) and monitoring approaches. The monitoring algorithm for Control-Flow Temporal Logic uses instrumentation information to improve efficiency.

This novel strategy of using information obtained during instrumentation to optimise the monitoring process introduces new perspectives of instrumentation and monitoring:

- Monitoring procedures have conventionally seen the program under scrutiny as a black box, so the monitoring procedures could not benefit from knowledge about the structure of the program. Our approach changes this and, assuming the presence of the source code of the program being monitored, sees each *event* in a trace as a structure that is linked precisely to the parts of source code that generated it.

- Instrumentation was always used as a way to filter the trace generated by the program, not to organise the monitor state. Our approach changes this by using instrumentation information in the various lookup operations required.

8.1.3 Offline Analysis

Once we had developed the new specification formalism along with its instrumentation and monitoring algorithms, it became clear from our main test case at CERN that we needed a way to determine possible causes of violations of specifications. This revealed two areas of Runtime Verification that were under-developed:

- Explaining failures to satisfy properties.
- Providing useful offline analysis tools.

These two categories are linked; the second, that of providing useful tools, boils down to the challenge of making any theory developed to explain failure accessible and usable to engineers.

A General Explanation Approach

Since Control-Flow Temporal Logic is low-level with respect to programs for which it is used to write specifications, our explanation approach exploits this. In particular, we focus on recording and comparing program paths that we record with a conservative amount of additional instrumentation. We also provide engineers with the ability to record values of key variables around concrete states, but we did not use this feature in our experiments.

Our work on explanation introduces a new approach to deal with many observed program paths, that is, a way to compare the paths using context-free grammars such that common points of divergence can be detected. We then extend this initial approach to the interprocedural setting to allow measurements of durations to be explained by key differences in paths taken by functions further down the call tree.

The machinery that we introduce for explanation is general. Our representation of paths requires only the symbolic control-flow graph, since paths are sequences of edges through such graphs. Further, our comparison approach requires only the context-free grammar that one can derive from a symbolic control-flow graph. Given this generality, our approach can be used in any situation where the symbolic control-flow graph of a program can be obtained (regardless of the specification language).

We have demonstrated the applicability of our explanation approach by integrating it with the semantics of Control-Flow Temporal Logic. That is, we considered program paths with respect to the concrete states and transitions that are found on them at runtime. In this context, our path comparison approach became a way to explain verdicts of properties expressed in Control-Flow Temporal Logic.

Providing Useful Tools

With path-based explanation, we have introduced an approach to translating this into something useful for engineers. Deciding how such a translation could work boils down to deciding

on the infrastructure in place to take the data obtained by VyPR at runtime (which has a non-trivial structure) to some representation that would help an engineer to decide which part of their code might be problematic.

The solutions we present in the scope of this thesis include a Python-based analysis library, which is used for all of the analyses shown in Chapter 7, and a web-based tool. The library wraps up all complex post-processing operations, such as path reconstruction and comparison, in functions that can be called by analysis scripts. The web tool is designed to be the entry point for analysing data generated by VyPR, with the library existing for analyses that cannot be performed with the web tool.

Ultimately, the weight of our contribution to explanation is found in the offline analysis tools provided by VyPR. While there is necessarily our theoretical contribution in the form of path-based explanation and its subsequent specialisation to Control-Flow Temporal Logic, the utility of the approach is only as useful as the tools that implement it. Hence our substantial contribution in this direction.

8.2 Application to Real-World Software

A common theme in this thesis has been the implementation of all of the theory developed so that application to real-world software is possible. Chapter 6 describes the implementation of the VyPR framework in a form that has been applied to multiple systems at CERN, with the most significant case study with the CMS Conditions Upload service being presented in Chapter 7.

Development of the VyPR framework has involved testing in multiple (real-world) environments, including:

- Web applications (written using the Flask framework and alternatives).
- Local, shorter-running Python programs.
- Unit-testing and Continuous Integration processes.

VyPR's monitoring architecture has remained stable in all work performed so far, working alongside a variety of other frameworks. Finally, the offline analysis facilities have been deployed in portable analysis notebooks (powered by Jupyter [jup]) that make analyses of real-world software we have included in publications reproducible.

8.3 Positioning in Existing Work

We now refer back to Chapter 2 in order to indicate the position of the contributions of this thesis in the existing literature.

8.3.1 Specification Languages

Our specification language, CFTL, was designed to have a low level of abstraction, to the point where additional information is not required to identify how events generated at runtime by

the program under scrutiny should link to the specification. We will now give a comparison between CFTL and various specification languages.

Temporal Logics

CFTL is a departure from the approaches taken by well-known specification languages such as Linear Temporal Logic [Pnu77] and Metric Temporal Logic [Koy90], which use atomic propositions alongside modal operators such as $\circ$ (next), $\Box$ (always) and $\diamond$ (eventually). Control-Flow Temporal logic dispenses with these modal operators and, while expressiveness is lost, the result is that the relatively straightforward properties that it was designed to express (for example, the ones we expressed for our case study in Chapter 7) can indeed be expressed without problems.

Further, in a departure from the conventional approach of the RV community (that of building all of the necessary expressive power into the specification language itself), we focused on developing analysis tools. Using these tools, any analyses that could not be performed solely with the use of CFTL specifications could be performed using CFTL specifications first, and then the analysis approaches that we developed in Chapters 5 and 6.

An existing specification language that moves towards expressing properties directly over events generated by source code is CaRet [AEM04], which augments LTL in order to describe constraints over calls and returns generated by the program under scrutiny. While this is indeed moving towards CFTL's level of abstraction, it 1) does not look at timing properties and 2) is fixed at the level of granularity of function calls (CFTL specifications talk about events generated on the level of individual statements in code).

We omit a comparison of CFTL with languages for concurrent systems (such as CTL [CE82] and CTL* [EH86]) simply because their semantics are defined over computation trees and their syntax provides features like *path quantifiers* in order to reason about multiple possible futures. The semantics of CFTL is defined in the setting where there is only one future, hence quantification over multiple possible futures makes little sense currently. The reasoning is the same for temporal logics concerning hyperproperties [CFK$^+$14]; the cases for which CFTL was developed do not have a need for properties to be defined over multiple traces at the same time, which is the setting in which the work on hyperproperties has been built up.

A Note on Truth Domains

Existing work [MP95, BLS10] has used the notion of a non-boolean truth domain when monitoring in the Runtime Verification context. While we have introduced a 3-valued truth domain for CFTL (see Chapter 5), the purpose of our truth domain is not the same as that of existing approaches. In particular, the motivation of existing work was to enable a monitoring mechanism to give a verdict before the program being monitored had finished. Our motivation, on the other hand, is simply to enable us to define the *falsifying concrete state* of a dynamic run once the monitored program has finished (ie, when we have a complete dynamic run).

Rules and Streams

We do not draw similarities between rule-based specification languages such as EAGLE [BGHS04] and CFTL, since one of the defining characteristics of EAGLE is the ability to use systems of rules to represent modal operators such as *always* and *eventually*, which CFTL avoids.

However, we do draw similarities between stream-based specification languages (such as Lola [DSS+05]) and CFTL. In particular, the implementation of VyPR can ultimately be said to collect *time series* (in the form of dynamic runs) of the data recorded in order to monitor for specifications and subsequently generate explanations of verdicts. Hence, one could draw a comparison between CFTL's operation over these time series and the operations enabled by languages such as Lola, however this link has not yet been explored and, given the practical motivation of work on CFTL and VyPR, it is not a priority.

Automata

We make a comparison between CFTL and the other automata-based methods (plans [BBKT05], QEA [BFH+12], DATEs [CPS09a]) simultaneously because the comparison is similar in each case. In particular, automata-based specifications tend to focus on *order-based* properties (one event should take place, and then another, and a bad event taking place will result in an error state). CFTL specifications, on the other hand, do not always impose an order on events. For example, consider the specification

$$\forall q \in \mathsf{changes}(x) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(g))) < 1 \wedge \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 2.$$

There is no constraint placed on the order in which the functions g and f should be observed; we only care about the time that each one takes. Hence, while some of the automata-based specification formalisms enable one to specify constraints over the timing behaviour of a system (DATEs [CPS09a]), the strong constraints placed on the order in which events occur is not a focus of CFTL.

Complex Event Processing

The application of Complex Event Processing to RV in the form of *BeepBeep* [Hal16] focuses on providing a formalism with which one can define sequences of operations to be performed on input data. While this does not directly relate to CFTL (because CFTL tends to perform quite simple processing on data observed during a system's runtime), Complex Event Processing could serve as inspiration for further development of the analysis tools that we developed in Chapter 6. For example, both our analysis library and web analysis environment focus on performing a sequence of transformations on data stored by VyPR in order to perform analyses. Such sequences of operations could be encoded in a declarative language for analyses.

8.3.2 Monitoring

While many existing approaches to monitoring based on temporal logics focus on the use of automata [VW94, GO01], the monitoring mechanism for CFTL (see Chapter 4) is formula trees.

Formula trees are uniquely suited to CFTL because of the lack of order enforced on the events that are observed at runtime. While constructing automata to monitor for CFTL specifications is possible, the automata would become large quickly, since all possible orders of observations would have to be encoded.

To conclude our brief discussion of how our monitoring algorithm fits into the existing literature, we highlight our approach for dealing with monitoring of universally quantified specifications. This problem is highlighted during the development of QEA [BFH+12, HRTZ18], and is dealt with by constructing indices over the monitor structure in order to optimise lookup. In this respect, the approach that we describe in Chapter 4 is similar.

8.3.3 Static Analysis and Instrumentation

A discussion of how our contributions fit into the static analysis and instrumentation landscapes requires us to look at the categories given in Section 2.9: instrumentation implementations, instrumentation languages, optimisation of monitoring via static analysis and code transformation.

Implementations of Instrumentation

In terms of instrumentation of Python, VyPR works at the bytecode level. In particular, it constructs the AST of the program under scrutiny, determines which statements in the program need to be instrumented, modifies the program AST and then compiles to bytecode. While there are many instrumentation approaches available for other languages (such as AspectJ [Asp] for Java and Cil for C [cil]), bytecode instrumentation for Python is not something that is done often, and so we had to implement this from scratch.

Instrumentation Languages

Given a specification written in a language with a high level of abstraction, such as LTL, a mapping must be given in order to define how the events occurring at runtime relate to the atomic propositions in the specification. A prominent example of this is the Java-MAC [KVK+04] framework, which allows definition of the specification, and then how runtime events relate to the specification.

Given the low level of abstraction of CFTL, this separate mapping is not needed. In fact, CFTL could be seen as *both* a specification language, and an instrumentation language, since specifications define both the property to be checked, and where to record the data to check it.

Residues

The notion of a *residue* (a specification that has been modified because part of it need not be monitored anymore after it was proven statically to hold) appears in different forms in both the STARVOORS [ACPS17] and the Clara [BLH10] projects. We highlight that the idea of attempting to prove part of a specification in order to reduce the overhead induced by monitoring could not be applied immediately to CFTL, since a prominent focus of CFTL specifications is timing properties. The timing behaviour of the system could be estimated (given an upper

bound) statically, opening the way for application of a static proof approach. However, one of the aims of an application of CFTL and VyPR, when analysing the performance of a program, is to find anomalous timing behaviour. This behaviour may not be encoded in a system model, meaning that any kind of proof involving the timing information of a system behaving in a certain way may result in violations being missed at runtime.

We do however highlight that CFTL specifications regarding the values held by variables would provide an opportunity for the modification of specifications based on the outcome of proofs attempted statically.

Finally, a significant characteristic of the Clara tool is the application of static analyses in order to remove instrumentation points, thus reducing the overhead induced on the program under scrutiny at runtime. Chapter 4 shows that our instrumentation approach for CFTL is similar, but there is one key distinction: Clara starts with an over approximation of the set of instrumentation points and progressively removes them, whereas our instrumentation approach starts with the empty set, and progressively constructs the set of instrumentation points by traversing a symbolic control-flow graph computed from the program. Further, since Clara works in the Java setting, it does more complex analyses of reference types (in order to monitor properties of certain objects across the entire system execution). VyPR does not currently attempt this because our use case in Chapter 7 did not have a need for it.

Code Transformation

We conclude our discussion of static analysis approaches with the final one from Section 2.9 that has immediate relevance to CFTL and VyPR: loop unrolling, a contribution by Dwyer et al. [DPP10]. This work is of particular significance to CFTL because it discusses transformations of the same structures that CFTL deals with.

The contribution from Dwyer et al. looks at monitoring with automata, and introduces the notion of the latest iteration of a loop that could have an impact on a monitoring result. Of interest to CFTL would be a special case of this: the unrolling of a single loop iteration in order to prevent repeat observations from a statement whose first execution is the only one of concern. For example, if the specification

$$\forall q \in \mathsf{changes}(x) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(f))) < 1$$

were to identify a call to f inside a loop, the instrumentation approach that we present in Chapter 4 would identify the statement that calls f as an instrumentation point. This would mean that, at runtime, the duration of every one of these calls to f would be measured, even though we only need the first one. Hence, in this case, unrolling the first iteration of the loop would lead to a reduction in overhead induced by monitoring.

This was not addressed in the thesis because unrolling the first iteration of a loop requires that we can guarantee that a certain path through the loop's body will be executed on the first iteration (if not, there may be another path on which a measurement should be taken and we would miss it). Making such a guarantee requires solving a Constraint Satisfaction Problem on the path conditions present inside the loop body. In Python, since variables are often not given

types statically, this problem would be difficult to solve. Given that our use case did not have a problem with high monitoring overhead because of loops, we chose not to address the problem.

8.3.4 Explanation

One of the main contributions of this thesis is the approach for explanation of property violations that we present in Chapters 5 and 6. We will now compare it to the areas of existing work that have the most relevance: path comparison and fault localisation.

Path Comparison

Existing work on path comparison [BL96, RBDL97] has, similarly to our contribution, focused on obtaining a program path that has enough detail to enable meaningful analyses to be performed. Our contribution is distinct in that we introduce a context free grammar-based approach that enables of determination of the regions of code over which multiple paths disagreed.

Fault Localisation

Our approach, based on collecting program path information and determining regions of code that may be responsible for false verdicts, can be compared to Fault Localisation if one considers program paths as the *spectra* that Spectrum-based Fault Localisation approaches consider. In particular, Spectrum-based Fault Localisation, such as that performed by tools like Tarantula [JHS01], tries to rank values of program spectra (methods, statements in code, etc) based on the frequency with which they appear in program runs generated by failing tests.

Our approach (an example of which is given in Section 7.7.3) is similar in that it replaces failing tests with failing verdicts (essentially, the test oracle is different), but then compares the spectra obtained at runtime (program paths) in order to isolate problematic regions of code.

8.3.5 Presentation of Explanation Results

Our contribution takes inspiration from many of the existing tools developed by the Software Engineering community that enable visual inspection of the performance data of a system [CLRG19, BMDR13, MMRL16, COL+17, RHV+09].

In particular, the development of our web based analysis environment (see Section 6.8) focused on presenting monitoring results in the context of the source code that generated them. In fact, given CFTL's low level of abstraction, it is straightforward to link measurements with the relevant parts of the specification and the relevant lines of code during offline analysis.

The key distinction between the existing contributions from the Software Engineering community and our analysis tools is that the results that we enable interaction with are generated by monitoring for a formal specification. The results visualised by existing tools are usually obtained by profiling of the program under scrutiny.

8.4 Future Work

The possibilities for future work are numerous and diverse. We group the possibilities that we see so far by the areas of research into which they fit.

8.4.1 Specification

A significant amount of development has taken place on Control-Flow Temporal Logic since the introductory paper, [DR19b], to the extent that the version of the logic given in this thesis is much more expressive than the original. This extension in itself gave rise to more ways to apply our explanation machinery.

Interprocedural Specifications

There are still clear opportunities for extension, the most notable being a departure from the single-scope setting to allow specification of interprocedural properties. So far any interprocedural analysis that we could perform was done offline, once data had been collected by monitoring multiple properties, each within a single scope. The most immediate way to allow interprocedural specifications is to consider a system with multiple procedures as a system containing multiple symbolic control-flow graphs. From here, we could extend the names of variables and functions commonly used in our specifications to be *fully-qualified*, that is, to also include the name of the function inside whose scope we wish to take a measurement.

Fully General Constraints

Given our underlying instrumentation and monitoring machinery, it would be feasible to extend the syntax (and therefore the instrumentation and monitoring) to support specifications such as

$$\forall c \in \mathsf{calls}(\texttt{calc}) : \mathsf{source}(c)(x) + \mathsf{source}(c)(y) \leq \mathsf{dest}(c)(z)$$

This was not done because, so far, we have no need to be able to include more than two quantities in a single atom, but the need may arise.

A Type System

Continuing with the theme of more general constraints, we consider that specifications in Control-Flow Temporal Logic deal with real state obtained during program runs. So far our work has focussed on adding more expressive power to how one can obtain the quantities to compare, but we have not addressed the problems that come with the rich set of types available in languages such as Python. For example, while our instrumentation approach is fully capable of obtaining the necessary information to decide whether one list is found inside another, we do not provide the syntax to do so. Further, we do not provide any way for engineers to perform type checking inside their specifications. For example, an ideal specification to check whether

an intermediate list is in a final list could be:

$$\forall q \in \mathsf{changes}(\texttt{someList}) : \\ q(\texttt{someList}) \text{ in } \mathsf{next}(q, \mathsf{changes}(\texttt{finalList}))(\texttt{finalList})$$

We could also add a primitive type check:

$$\forall q \in \mathsf{changes}(\texttt{someList}) : \\ \mathsf{type}(q(\texttt{someList})) = \mathsf{list} \implies q(\texttt{someList}) \text{ in } \mathsf{next}(q, \mathsf{changes}(\texttt{finalList}))(\texttt{finalList})$$

However, this would have the shortcoming that our monitoring procedure would attempt to decide whether `q(someList)` is *contained within* `next(q, changes(finalList))(finalList)` if the type of `q(someList)` were found to not be list.

Hence, type checking would need to be implemented on the level of the monitoring algorithm (it cannot be performed statically during instrumentation since types cannot always be statically determined in Python), and the syntax that we provide would need to be extended.

Existential Quantification

So far, we have observed no need for existentially quantified specifications, but the extension of Control-Flow Temporal Logic to support such a structure would be interesting. For example, considering our *hash checking effectiveness* study (in which we investigated how often hash checking determines that at least 70% of proposed Payloads must be uploaded), we may wish to assert that, given an assignment to the list `hashes`, there are at least 70% of such hashes that are not found. One could imagine using existential quantification as such:

$$\forall q \in \mathsf{changes}(\texttt{hashes}) : \\ \exists_{\geq 0.7 \times \mathsf{length}(q(\texttt{hashes}))} c \in \mathsf{future}(q, \mathsf{calls}(\texttt{check})) : \\ \mathsf{result}(c)(\texttt{found}) = \texttt{False}$$

However the real motivation to include existential quantification is so far weak, so this direction remains a low priority.

8.4.2 Instrumentation

We highlight two directions for future work in instrumentation. Both are based on the problems that one encounters when performing instrumentation for a weakly-typed language like Python.

Data-flow Analysis for Reference Variables

Our first proposed direction is that of using data-flow analysis to improve the instrumentation of variables with reference types. Currently, software engineers can tell VyPR whether a variable is a reference type or not and the instrumentation will subsequently be adjusted. However, our current implementation is an over-approximation, that is, we are likely to instrument too many program points.

A proposed solution is to employ some static analysis of the data-flow of the program under scrutiny to decide which program points can *definitely* be thrown out of the set of instrumentation points.

Dynamic Changes to Instrumentation

Variables in Python have the ability to change their types (between primitive and reference) at runtime. If one is trying to make instrumentation as precise and conservative as possible, this causes problems. In our current instrumentation approach, we do not address this fact since we have never had reason to, but use cases may arise where this is needed.

Our proposed solution so far would be to provide a custom Python interpreter that can add and remove instrumentation code from the sequence of bytecode instructions that the normal C-based Python interpreter processes.

8.4.3 Offline Analysis

The proposed research directions for offline analysis are into ways to make the analysis more efficient, especially when there is a lot of data. As shown in Chapter 7, when post-processing involves steps such as call tree construction and path comparison, for large amounts of data the time taken for this processing can be large.

A Declarative Analysis Language

As seen in Chapter 7, the effort required for more complex analyses using VyPR's analysis library is still significant. While we have made some progress in our development of the web-based analysis environment (which enables fast execution of some fixed types of analyses), the next step would be to introduce a declarative language in which one could describe their analysis, and the existing analysis tools would be used in order to generate a result.

Optimising Path Comparison

Path comparison requires intersection of the *parse trees* that we derive using a combination of program paths and context-free grammars. Since this intersection is commutative, it would be possible to store intermediate results. This means that we could implement some intersection process on the verdict server to run at regular intervals, as data is received from monitoring. This would mean that, for example, instead of performing the intersection across 4000 paths, we intersect the result from the first 3000 and then perform a subsequent intersection over the result with the remaining 1000 paths.

The exact policy adopted as to when to perform such intermediate computations would be a subject of research; one could hope that such timing could be automatically determined.

Call Tree Construction

Our current approach to this is to use the timing information associated with the function calls that VyPR stores. This can result in many (quite inefficient) queries and strong dependence of the post-processing time on the size of the call tree (seen in Chapter 7).

Our proposal would be to construct a call stack at runtime and store the relationships encoded in this stack with the monitoring data. There would still be some post-processing required for constructing call trees using data from multiple machines, but all calls from the same machines would be linked without any post-processing.

Paths with Timing

It can occur that all of the paths in a set being used for an intersection are the same and no path parameters are obtained by intersection. In this case, it would be reasonable to ask what the differences in timing were. Since, with our current instrumentation only recording path information, timing information is not available, so the only clear solution would be to insert time measurement alongside each branch recording instrument and attach this timing information to the parse trees that we derive. A direction of the research from here could be how to compare this new structure (a parse tree combined with timing information) so as to be able to automatically detect strange timing behaviour in paths recorded across a period of time.

8.4.4 Closing Remarks

The work described in this thesis will be continued as a project at CERN, where it will be used (and maintained) as a program analysis framework for software employed at the CMS Experiment.

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter Schmitt. The key tool. *Software & Systems Modeling*, 4:32–54, 02 2005.

[ABM98] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241)*, pages 46–54, 1998.

[ACP20] Shaun Azzopardi, Christian Colombo, and Gordon Pace. Clarva: Model-based residual verification of java programs. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, 2020.

[ACPS17] Wolfgang Ahrendt, Jesus Chimento, Gordon Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, 51, 08 2017.

[AEM04] Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. A temporal logic of nested calls and returns. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481. Springer, 2004.

[AH93] R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35 – 77, 1993.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181?203, January 1994.

[APS12] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. A unified approach for static and runtime verification: Framework and applications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 312–326, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Asp] AspectJ. `https://www.eclipse.org/aspectj/`.

[AZG06] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th*

Pacific Rim International Symposium on Dependable Computing, PRDC '06, page 39?46, USA, 2006. IEEE Computer Society.

[AZGvG09] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780?1792, November 2009.

[BBDC+12] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, Feb 2012.

[BBKT05] Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing conformance of real-time applications by automatic generation of observers. *Electronic Notes in Theoretical Computer Science*, 113:23 – 43, 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

[BBL+20] Ezio Bartocci, Luca Bortolussi, Michele Loreti, Laura Nenzi, and Simone Silvetti. Moonlight: A lightweight tool for monitoring spatio-temporal properties. In Jyotirmoy Deshmukh and Dejan Ničković, editors, *Runtime Verification*, pages 417–428, Cham, 2020. Springer International Publishing.

[BCBC+18] Ansem Ben Cheikh, Yoann Blein, Salim Chehida, German Vega, Yves Ledru, and Lydie du Bousquet. An environment for the partrap trace property language (tool demonstration). In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 437–446, Cham, 2018. Springer International Publishing.

[BCE+14] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 31–47, Cham, 2014. Springer International Publishing.

[BFB+17] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, 2017.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification. Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, February 2018.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 44–57, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BGK+13] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. Adaptive runtime verification. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, pages 168–182, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BHL+07] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. In Oleg Sokolsky and Serdar Taşıran, editors, *Runtime Verification*, pages 22–37, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[BLH10] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 183–197, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BLH12] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Trans. Program. Lang. Syst.*, 34(2), June 2012.

[BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. *J. Log. Comput.*, 20:651–674, 06 2010.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[BMDR13] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 63–72, 2013.

[BMM+19] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. Automatic failure explanation in cps models. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 69–86, Cham, 2019. Springer International Publishing.

[BNF11] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, pages 88–102, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Büc90] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[cer] OpenStack. `https://openstack.cern.ch/`.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 265–284, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CHM+19] Maria Christakis, Matthias Heizmann, Muhammad Numair Mansur, Christian Schilling, and Valentin Wüstholz. Semantic fault localization and suspiciousness ranking. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 226–243, Cham, 2019. Springer International Publishing.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[cil] Cil. `https://cil-project.github.io/cil/`.

[CLRG19] J. Cito, P. Leitner, M. Rinard, and H. C. Gall. Interactive production performance feedback in the ide. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 971–981, 2019.

[CMS16] CMSDBServiceStats. `http://jdawes.web.cern.ch/jdawes/talks/perf-meas-demo.pdf`, 2016.

[Col08] The CMS Collaboration. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[COL+17] J. Cito, F. Oliveira, P. Leitner, P. Nagpurkar, and H. C. Gall. Context-based analytics - establishing explicit links between runtime traces and source code. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 193–202, 2017.

[CP18] Christian Colombo and Gordon J. Pace. Considering academia-industry projects meta-characteristics in runtime verification design. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 32–41, Cham, 2018. Springer International Publishing.

[CPAM09] Christian Colombo, Gordon Pace, Patrick Abela, and Ixaris Malta. Offline runtime verification with real-time properties: A case study. 01 2009.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, pages 135–149, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37, Nov 2009.

[cpy] CPython. `https://github.com/python/cpython`.

[Daw17] Joshua H Dawes. A python object-oriented framework for the CMS alignment and calibration data. *Journal of Physics: Conference Series*, 898:042059, oct 2017.

[dH05a] Marcelo d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *WODA '05*, 2005.

[dH05b] Marcelo d'Amorim and K. Havelund. Jeagle: a java runtime verification tool. 2005.

[DHJ+20] Joshua Heneage Dawes, Marta Han, Omar Javed, Giles Reger, Giovanni Franzoni, and Andreas Pfeiffer. Analysing the Performance of Python-based Web Services with the VyPR Framework. In *Runtime Verification, Los Angeles, USA, 2020*, 2020.

[DHR+19] Joshua Heneage Dawes, Marta Han, Giles Reger, Giovanni Franzoni, and Andreas Pfeiffer. Analysis Tools for the VyPR Framework for Python. In *International Conference on Computing in High Energy and Nuclear Physics, Adelaide, Australia 2019*, 2019.

[DLZ05] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, AADEBUG'05, page 99?104, New York, NY, USA, 2005. Association for Computing Machinery.

[DM10] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, pages 92–106, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[DPP10] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis? In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 36–50, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[DR19a] Joshua Heneage Dawes and Giles Reger. Explaining Violations of Properties in Control-Flow Temporal Logic. In *Runtime Verification, Porto, Portugal 2019*, 2019.

[DR19b] Joshua Heneage Dawes and Giles Reger. Specification of temporal properties of functions for runtime verification. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*, pages 2206–2214, 2019.

[DRF+19] Joshua Heneage Dawes, Giles Reger, Giovanni Franzoni, Andreas Pfeiffer, and Giacomo Govi. VyPR2: A Framework for Runtime Verification of Python Web Services. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, pages 98–114, 2019.

[dSCK16] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005.

[dyn79] Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[EB08] Lyndon Evans and Philip Bryant. LHC machine. *Journal of Instrumentation*, 3(08):S08001, 2008.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *J. ACM*, 33(1):151?178, January 1986.

[FKRT18] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 241–262, Cham, 2018. Springer International Publishing.

[FMN15] Thomas Ferrère, Oded Maler, and Dejan Ničković. Trace diagnostics using temporal implicants. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, pages 241–258, Cham, 2015. Springer International Publishing.

[GCKS06] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, Jun 2006.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[GRM+18] Mojdeh Golagha, Abu Mohammed Raisuddin, Lennart Mittag, Dominik Hellhake, and Alexander Pretschner. Aletheia: A failure diagnosis toolchain. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 13?16, New York, NY, USA, 2018. Association for Computing Machinery.

[Hal16] Sylvain Hallé. When rv meets cep. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 68–91, Cham, 2016. Springer International Publishing.

[Hav08] Klaus Havelund. Runtime verification of c programs. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, pages 7–22, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[HOW14] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 178–192, Cham, 2014. Springer International Publishing.

[HRTZ18] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. *Monitoring Events that Carry Data*, pages 61–102. Springer International Publishing, Cham, 2018.

[JDH+20] Omar Javed, Joshua Heneage Dawes, Marta Han, Giles Reger, Giovanni Franzoni, Andreas Pfeiffer, and Walter Binder. PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects. In *Automated Software Engineering, Melbourne, Australia, 2020*, 2020.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273?282, New York, NY, USA, 2005. Association for Computing Machinery.

[JHS01] James Jones, Mary Harrold, and John Stasko. Visualization for fault localization. 08 2001.

[JMLR12] Dongyun Jin, Patrick Meredith, Choonghwan Lee, and Grigore Rosu. Javamop: Efficient parametric runtime monitoring framework. *Proceedings - International Conference on Software Engineering*, pages 1427–1430, 06 2012.

[jup] Jupyter. `https://jupyter.org`.

[Kam68] Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Ucla, 1968.

[KBC16] Denis Kuperberg, Julien Brunel, and David Chemouil. On finite domains in first-order linear temporal logic. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis*, pages 211–226, Cham, 2016. Springer International Publishing.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, Nov 1990.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[KVK+04] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24:129–155, 03 2004.

[MMRL16] R. Minelli, A. Mocci, R. Robbes, and M. Lanza. Taming the ide with fine-grained interaction data. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, Los Alamitos, CA, USA, may 2016. IEEE Computer Society.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.

[PCD+06] I Papadopoulos, R Chytracek, D Dullmann, G Govi, Y Shapiro, and Z Xie. CORAL, A Software System for Vendor-Neutral Access to Relational Databases. 2006.

[PCJ+17] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620, 2017.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[pyta] Extending Python with C. `https://docs.python.org/2.7/extending/extending.html`.

[pytb] Flask Framework for Python. `https://flask.palletsprojects.com/en/1.1.x/`.

[pytc] Python settrace. `https://docs.python.org/2/library/sys.html#sys.settrace`.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering — ESEC/FSE'97*, pages 432–449, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[RCE+19] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. The city metaphor in software visualization: Feelings, emotions, and thinking. *Multimedia Tools and Applications*, 05 2019.

[Reg15] Giles Reger. Suggesting edits to explain failing traces. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 287–293, 2015.

[Reg17] Giles Reger. A report of rv-cubes 2017. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 1–9. EasyChair, 2017.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57 – 95, 1987.

[RH01] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*, pages 83–91, 2001.

[RHV+09] David Rothlisberger, Marcel Harry, Alex Villazon, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting static source views in ides with dynamic metrics. pages 253–262, 09 2009.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 30–39, Piscataway, NJ, USA, 2003. IEEE Press.

[sim] Simulink. `https://www.mathworks.com/products/simulink.html`.

[SKF20] Chukri Soueidi, Ali Kassem, and Yliès Falcone. Bism: Bytecode-level instrumentation for software monitoring. In Jyotirmoy Deshmukh and Dejan Ničković, editors, *Runtime Verification*, pages 323–335, Cham, 2020. Springer International Publishing.

[sql] SQLAlchemy. `https://www.sqlalchemy.org`.

[SSA+19] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Sr?an Krstić, Joa?o M. Lourenço, Dejan Nickovic, Gordon J. Pace, Jose Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods in System Design*, 54(3):279–335, 2019.

[sta] `pyinstrument`. `https://github.com/joerick/pyinstrument`.

[tie] Tier0. `https://home.cern/science/computing/grid-system-tiers`.

[vue] VUE.JS. `https://vuejs.org`.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1 – 37, 1994.

[ZBK20] Eleni Zapridou, Ezio Bartocci, and Panagiotis Katsaros. Runtime verification of autonomous driving systems in carla. In Jyotirmoy Deshmukh and Dejan Ničković, editors, *Runtime Verification*, pages 172–183, Cham, 2020. Springer International Publishing.

[ZELS19] Teng Zhang, Greg Eakman, Insup Lee, and Oleg Sokolsky. Overhead-aware deployment of runtime monitors. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 375–381, Cham, 2019. Springer International Publishing.

[ZJLS16] Lukasz Ziarek, Bharat Jayaraman, Demian Lessa, and J. Swaminathan. Runtime visualization and verification in jive. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 493–497, Cham, 2016. Springer International Publishing.