Failure Detection and Diagnosis in Architecture-based Autonomic Systems

Paulo Casanova

CMU-S3D-23-100

April 2023

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Garlan (Co-chair, Carnegie Mellon University)
Mário Zenha-Rela (Co-chair, Universidade de Coimbra)
Jonathan Aldrich
Claire Le Goues
Rui Abreu (Universidade do Porto)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

Abstract

As the size and complexity of modern IT systems increases, there is greater need for automatic recovery from failures. Recently, self-adaptive control loops have started to replace human oversight as means to ensure high availability of software systems. Two critical pieces of the self-adaptive loop for high availability are failure identification and fault localization. Failure identification – figuring out that something is not working – is a challenging activity as (1) the monitoring is not done at the same abstraction level as the failures manifest themselves, and (2) because systems perform several activities concurrently, incorrect behavior will appear mixed with correct behavior. Identifying faults, pinpointing the source of the failure, is also challenging as (1) there may be multiple explanations for a fault and (2) diagnosis must be performed in a useful time frame. In this thesis, we propose to improve self-diagnosis through a framework that allows a system to identify failures and pinpoint the corresponding faulty parts in a running system. This framework is based on two key principles: reasoning about the system's behavior at the software architecture level and providing a declarative approach to describe system behavior. The use of architectural models allows the diagnostic infrastructure to scale gracefully, supports efficient run-time execution of common fault localization algorithms, and supports failure diagnosis of system-level properties such as end-to-end performance. The use of a declarative approach to behavior allows one to systematically specify rules for bridging the gap between low-level monitoring and higher-level problem detection. It also supports reuse across systems that share a common architectural style or implementation infrastructure.

To my adviser, without whom I would never have started and finished my PhD.
To my parents, who provided me with the love, care and knowledge that has guided my life.
To my children, for whom I do everything, including my Thesis.
And, above all, to my wife, who's unbounded love and unmatched support has given me the strength to achieve all thus far.

Acknowledgments

This work is co-financed by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's complex computer systems form the significant core of many businesses. For many of these systems, an increasingly important characteristic is high availability, as downtime is extremely costly. For example, Google's advertising revenues for 2022 were around 224B: this means that downtime has a direct cost of more than 1K$ per second. However, in Google's data centers, thousands of servers fail over a year, sometimes many dozens at the same time [56]. Still, Google serves over 10 billion searches every month in the United States with almost no downtime [77]. Google is an extreme example, but many other companies like Netflix or entire industries such as banking or air transportation share similar requirements. Even simple services from a mobile application, such as authentication, may require high availability for some businesses.

Preventing downtime in these systems is a challenging task. Modern systems are built from hundreds, or even thousands, of components that interact with each other in many different ways. Many of these components are independently developed by third-parties: some are off-the-shelf, some are custom-built and some are even upgraded without control of the owner of the system. Many components will eventually fail unpredictably and many will be subject to unexpected loads. And, many components may be subject to an open-ended set of internal and external cyber attacks.

The traditional way to address availability, still widely used, relies on human oversight. While humans provide a great deal of flexibility and adaptability, they are expensive, slow to process information, slow to react, unpredictable and prone to disastrous mistakes. Two high profile examples of failures due to human error was the Amazon's cloud services unavailability in 2011 and 2017 and Meta's massive outage in 2021. In the all cases incorrect change of network configuration by a human administrator ultimately led to the failure of the entire data centers for Amazon [72, 73] and the entire Facebook, WhatsApp, Instagram and Oculus services for Meta [51].

A different, complementary, technique, also widespread, is the use of local recovery mechanisms such as timeouts, load balancing and retries. For example, Google's file system [34] is a system in which this technique is used extensively: system components are designed assuming frequent communication and hardware failures. These automatic solutions avoid having humans manage the system, but need to be built into the system itself, which is not always possible. Also, they are expensive to develop as they require a significant amount of handcrafted code and they

increase the system's complexity. And because they are local, they don't always produce the appropriate systemic (non-localized) behavior. For example, Netflix's internal system flooding due to the system's own retry mechanisms in 2011 [7] shows how local recovery may, in fact, result in increased system downtime. In Netflix's case, a slowdown in a group of servers triggered a flood of retries from the dependent servers, starting a feedback spiral that ultimately led to service unavailability. These failures are so common that Google published document a few years later about this [64].

For complex systems that require high availability in spite of component failures and environmental unpredictability, a different approach can be taken that, in a way, leverages the strengths of both traditional approaches – human oversight and automated recovery – reducing their weaknesses. This approach consists in using a second system that monitors and adapts the main system in response to failures and environmental changes: it mimics humans by externally observing and changing the system, but it is fully automated. This approach is generally known as *autonomic computing* [45], and systems following this approach are often referred to as *self-adaptive systems* [20, 28].

Autonomic computing is based on a form of closed-loop control in which a *base system* provides the business-related services and a *controller* system monitors the base system, and, by monitoring its behavior, decides *when* to perform changes and *what* to change. In this scenario, the controller has to perform a set of four top-level activities which naturally follow from its responsibility. These activities consist in *monitoring* the system, *analyzing* the monitored data to detect the need for adaptation, *planning* how to adapt and *executing* the plan. These activities are depicted in figure 1.1 in what is sometimes referred to as the *MAPE loop* [39].

A considerable amount of research has been done in autonomic computing over the last decade [20, 28]. While significant research has been performed in the areas of planning and execution, the *monitoring and analysis* parts of the MAPE loop have received significantly less attention. Here, three activities become critical in the design of this stage: (1) *deciding what to monitor* – choosing where to look, (2) *identifying failures* – figuring out that something is not working properly; and (3) *identifying faults* – determining what are the sources of those failures.

Deciding what to monitor in a system is a complex task, whether designing a new probing infrastructure or reusing an existing one. Designing a probing infrastructure forces a tradeoff be-

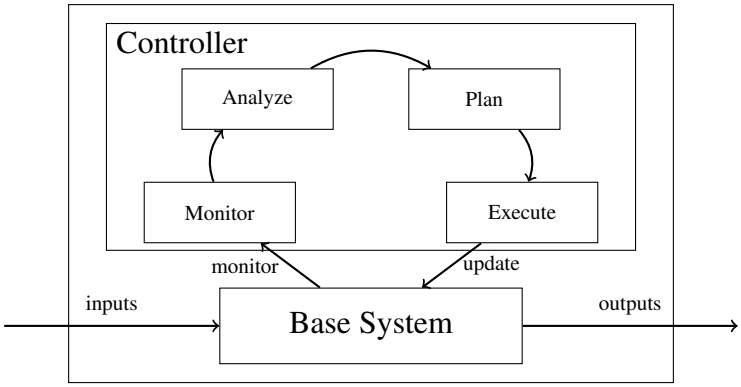


Figure 1.1: The MAPE loop.

tween monitoring cost and the benefits of the observation. For example, monitoring the latency of a web server may introduce an additional point of failure, reducing system availability. Using an existing probing infrastructure forces a discussion on *adequacy*: can the existing probes provide enough information to feed the MAPE loop?

Identifying failures in a running system is challenging. First, while failures commonly manifest themselves at a high level of abstraction, like degradation of quality of service below minimum thresholds or violation of high-level protocols, monitoring is usually performed at a lower level. Hence, we need to bridge the gap between these two levels. For example, two of the top 2021 web attack categories, identification and authentication failures (number 7) and injection (number 3) [58], appear to be correct behavior when solely observing their corresponding low-level network traffic. Only by analyzing higher level interactions between users and the systems can we identify these attacks. Second, today's systems perform multiple activities concurrently. These activities, even if they are unrelated to each other, will appear mixed to an observer. For example, an observer of a web server will notice multiple file open and close operations, which are not easily related to client requests that are currently being served.

Identifying faults is also challenging, even assuming that failures are correctly identified. First, there can be multiple possible explanations for a failure. For example, a system slowdown may be caused by a slow database, a denial of service attack, or a faulty network connection. Second, identification has to be performed within a useful time frame. For example, if identification takes too long to detect a denial of service attack, system availability may be compromised.

Given the difficulty of addressing these challenges, detection and diagnosis are typically associated with high development cost, low quality or both. Cost arises from the large amount of special-purpose and handcrafted code that has to be produced from scratch for each system. For example, there are thousands of lines of code in Hadoop just to detect server failures. Low quality arises from the locality of diagnosis which disregards systemic properties and emergent behavior, from the inability to ensure coverage of all behavior that may signal failures, and from the lack of efficient but accurate algorithms that systematically determine and order the possible causes of the detected failures.

In this thesis we present an approach that simultaneously improves diagnosis quality and also lowers its cost by introducing a diagnostic *framework*[1] that can be easily instantiated for specific systems. This framework is based on two key principles. The first principle is *reasoning about the system's behavior at the software architecture level* instead of the code level. As we will argue, because architectural models provide a high-level perspective on system behavior, architecture models simplify the reasoning process for fault diagnosis, scale to large and complex systems, support the detection of systemic (non-localized) faults, and provide a foundation to reason about repair. The second principle is *providing a declarative approach to describe system behavior*, including failure conditions, reducing the cost of specifying the behavior of individual systems and providing specification reuse, where general behavioral descriptions can be specialized to individual systems and then analyzed for properties of interest. With this approach, systems can be built and analyzed incrementally, reusing previous systems' behavior specifications.

[1] By framework we mean a library of reusable code, a set of tools, algorithms and design principles.

Thesis Statement

It is possible to lower cost while improving the quality of autonomic diagnosis by providing a general, scalable and practical framework that can identify failures and map them to their sources at the software architecture level.

The key claims in this statement involve cost, quality, generality, scalability and practicality. Let us expand each of these individual claims.

Cost: The framework will improve *cost* with respect to:

1. *Code Reuse*. Through the introduction of reasoning about behavior at different abstraction levels and the reuse of probes and behavioral patterns associated with architectural styles, much of the development can be transferred from project to project. Although the amount of reuse will vary from project to project, for systems that use common styles, we achieve over 50% reuse of the diagnosis code.
2. *Probing Costs*. The framework will reduce run-time costs by providing reasoning about unobserved behavior and supporting dynamic reconfiguration of probe deployment. Reasoning about unobserved behavior allows reducing the number of deployed probes that are needed to monitor the system. Reasoning about dynamic probe placement – adjusting which probes are deployed depending on context – allows even further optimization deploying probes only when needed.

Quality: The framework will have *higher quality* than an equivalent handcrafted code for a specific system with respect to:

1. *Systemic property support*. Reasoning about diagnosis in the framework allows taking systemic properties, such as performance or availability, into consideration.
2. *Guarantee of behavior coverage*. The framework will provide a perfect behavior classifier, with respect to the provided behavior model. All incorrect behavior, and only incorrect behavior, will be flagged as such.
3. *Systematic finding and ordering of failure cases*. The framework will provide a well-defined, systematic fault identification and ordering algorithm.
4. *Probe adequacy*. The framework detects probes whose output is not needed for diagnosis and detects missing probes in the system.
5. *Localization algorithms*. The framework ships with a localization algorithm but allows other algorithms to be plugged in, provided they can operate on the same set of observed data.

Generality: The framework will be *general* with respect to four dimensions:

1. *Style of system*. It will work with a range of different domains such as web systems, databases, and distributed systems.
2. *Problem classes*. It will detect failures both in functional requirements and quality attributes such as performance, security, and dependability.
3. *Implementation technology*. The framework will be applicable to systems developed in Java, C, and other languages. It will be independent of the platform in which the system

executes.

Scalability: The framework will be *scalable* with respect to three dimensions:

1. *Size of the system*: The framework will be capable of performing diagnosis in systems with possibly thousands of components.
2. *Monitoring data volume*: The framework will be capable of performing diagnosis in systems with more than one thousand monitored communication events per second.
3. *Monitoring overhead*: The framework will impose low monitoring overhead in the base system.

Practicality: The framework will be *practical* insofar as designers of autonomic systems will be able to use it without requiring fundamental knowledge other than standard state-of-the-practice in software engineering. Further, by supporting incremental construction, the framework will provide incremental diagnosis in complex systems as software developers will not need to build the whole diagnosis system in a single cycle.

Contributions

The contributions of this work are:

- A formalization of system behavior in terms of its software architecture that allows autonomic diagnosis to be performed;
- A technique for inferring high-level system behavior from monitored low-level behavior;
- A formalization of correct behavior that spans a wide variety of quality attributes including performance, availability and security;
- A technique for performing diagnosis applicable at run time capable of accurately inferring the sources of a failure;
- A technique for dynamically adding and removing probes that balances deployment cost and monitoring overhead with changing needs to observe system behavior;
- Metrics to measure diagnosis accuracy.

Publications resulting from this research include:

- Paulo Casanova, Bradley R. Schmerl, David Garlan, and Rui Abreu. *Architecture-based run-time fault diagnosis*. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, ECSA, volume 6903 of Lecture Notes in Computer Science, pages 261–277. Springer, 2011. ISBN 978-3-642-23797-3. [16]
- Paulo Casanova, David Garlan, Bradley Schmerl and Rui Abreu. *Diagnosing architectural run-time failures*. In Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 20-21 May 2013. Received SEAMS 2013 Best Paper Award. [17]
- Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. *Diagnosing unobserved components in self-adaptive systems*. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pages 75–84, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2864-7. [19]

- Paulo Casanova, Bradley Schmerl, David Garlan, Rui Abreu and Jungsik Ahn. *Applying Autonomic Diagnosis at Samsung Electronics*. Technical report, CMU-ISR-13-111, Institute for Software Research, Carnegie Mellon University, September 2013. [18]

Thesis Layout

The remainder of the thesis is organized as follows:

- Chapter 2 presents an example system used to illustrate the problems and identify the associated research questions addressed in this work.
- Chapter 3 describes, at a high-level, the approach and main principles behind it. Some examples of the framework use are presented in this section, although not much detail is provided.
- Chapter 4, Chapter 5 and Chapter 6 describe the CALL language, its semantics and its runtime.
- Chapter 7 describes other work that is complimentary or alternative to the one provided in this thesis.
- Chapter 8 discusses how the claims established in Chapter 1 are validated.
- Chapter 9 contains a discussion on this work and its consequences, provides detail on its limitations, adoptability and provides pointers for future work.

Chapter 2

Motivation and research challenges

In this chapter, we present an example of a representative system that will be used throughout the rest of the document to illustrate the research challenges and our approach to addressing them.

Consider an example of a standard high-load web system whose architecture is depicted in figure 2.1. In this system, a number of clients connect to one or more dispatchers, which forward requests to web servers connecting to a database. Dispatchers, also known as load balancers, forward requests to the web servers, which may query the database to obtain data to build a response. Dispatchers pick servers in a round-robin fashion, but respect client sessions by forwarding all requests from the same client to the same server.

In such a system a variety of failures can occur: web servers can stop responding, the database may slow down, network connections may fail intermittently leading to errors or delays, the dispatchers may forward requests to the wrong web server forcing a client to lose its session and its data, images may be missing from the file system, clients may request non-existing web pages or malicious clients may attack the system by attempting a denial of service. A more complex failure is a client being subject to a cross-site request forgery attack by another web site trying to steal the client's stored data in this system.

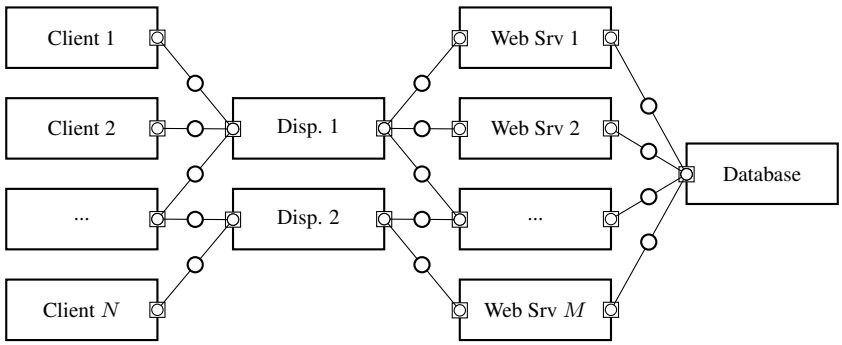


Figure 2.1: Example web system.

2.1 Detecting failures

Suppose a client performs a request to the system of figure 2.1 and receives a response from the web server: how does one know whether this is correct behavior? It certainly matches the expected observation in a web system from a high-level viewpoint insofar as a request was followed by a response. But is this enough to ensure no failures have occurred?

We know, from the HTTP protocol standard, that a response code of `500` represents a failure to process the client's request. But was it the computation that failed? Was it the request to the web server? The internal server processing of the request? The whole request/response?

Defining which computations have failed requires identifying which types of computations are performed by a system and defining under which conditions we know that an individual computation has failed.

> ? **Research Questions**
> | **RQ 1** How to identify which types of computations are performed by the system?

Given the complexity of modern software systems, it cannot be expected that system designers will provide a way to model all possible types of computations in a system. But, in practice, software systems reuse many concepts and structures: for example, `500` response codes represent a failure in *any* web system. A practical framework for diagnosis will require system designers to only describe the computations that are specific to the system being designed. More general computations associated with the system's architectural style can be reused.

> ? **Research Questions**
> | **RQ 2** How to reuse types of computations and correctness criteria in systems that share concepts and structures?

2.2 Identifying finite computations

The system in figure 2.1, like most interesting systems, has *infinite behavior*: there is no finite limit to the number of requests that it can handle, nor its processing time. But while this system as a whole has infinite behavior, the system's architects would hardly see it as performing a single, infinite computation.

The system's activity can be naturally decomposed into an infinite number of finite computations. One such decomposition is to split the system's behavior into individual clients requests. We can then, in principle, consider the correctness of each request in isolation.

However, isolating client requests in the system in figure 2.1 is not the only way to split computations that occur in the system. In web systems clients perform several requests enclosed in a *session*. While individual requests can be used to evaluate quality attributes such as response time, sessions can be used to evaluate quality attributes such as security. For example, a cross-site

request forgery may only be detectable through the observation of several requests in the same session.

In general, it will be necessary to perform multiple decompositions of the system's activity into individual, finite computations. Different correctness criteria can then be applied to different decompositions.

To handle research question 1 (identifying computation types) we need to be able to decompose the system's infinite behavior into individual computations.

> ? **Research Questions**
> **RQ 3** How to decompose the system's infinite behavior into the finite computations for which correctness criteria can be defined?

In the system of figure 2.1, many clients access the system in parallel and, therefore, several individual, finite computations are being processed simultaneously, spread across the components of the system. Applying correctness criteria to the computations requires being able to identify each computation individually.

> ? **Research Questions**
> **RQ 4** How to identify individual computations for a system that is executing many of them concurrently?

2.3 Mapping failures to their sources

Even if we have a solution for research questions 3 (decomposing infinite behavior) and 1 (identifying computation types), we still need to be able to map the observed failure to the source elements that are responsible. This is termed "fault localization."

If a client makes a request and the computation fails, is it a problem in the client? In the dispatcher? In the web server? This identification becomes hard in the presence of multiple, possibly correlated, faults. For example, a faulty server configuration may lead to decreased performance only when a certain dispatcher is used with a certain web server. (See Appendix A for more details on correlated faults.)

Although fault localization at *run time* remains an open research challenge, it has been tackled with considerable success at *design time*. For example, at design time, individual unit tests can be executed, one at the time, in an instrumented system. The list of executed instructions or program blocks, together with the result of the unit tests (success of failure), is fed into a fault localization algorithm, which pinpoints the possible locations of the failure. (See section 7.2 for more details on these algorithms.)

The success of design-time solutions suggests the possibility of using them at run time. However, several difficulties arise when adapting design-time fault localization algorithms to run time. Two difficulties are embodied in research questions 3 (decomposing infinite behavior) and 4

(identifying individual computations): when executing an individual unit test, the system behavior is finite and comprises a single individual computation that can be evaluated independently of other unit tests.

Another difficulty that arises is the balance between the time taken to produce a diagnosis and its respective accuracy. Since fault localization is typically performed using statistical methods (as we will see later), the more data collected, the more accurate the output will be. At design time – the target for the fault localization algorithms – the time it takes to produce a diagnosis is mostly irrelevant. But at run time it may be a critical factor. The time required to gather enough information to produce an accurate diagnosis may render the diagnosis useless, as too much harm may have already been done. Depending on the concrete situation, it may be preferable to act faster, but with less information and, therefore, with a less accurate diagnosis.

? Research Questions

RQ 5 Can we adapt design-time algorithms for fault localization to run time?

RQ 6 How to balance diagnosis accuracy against diagnosis performance and system impact?

2.4 Probing and diagnosis accuracy

In the previous subsection we mentioned the tradeoff between diagnosis accuracy and diagnosis performance (in terms of the time to perform a diagnosis). However, diagnosis accuracy is itself limited by probing. For example, if we cannot place probes in the connectors between the web servers and the database, we cannot know when the web server has accessed the database. While for some systems it may be possible to place probes wherever they are needed, sometimes other concerns may prevent probes from being placed: for example, technical limitations on the probe technology, or the risk that probes may interfere with system behavior.

Even in cases where probes can in principle be deployed, it may not be desirable to do so. For example, probes may decrease performance and they may also negatively affect other quality attributes such as availability (because they add an additional point of failure), or security (as they may introduce new vulnerabilities). Because probes may have a negative impact on some system properties, it may, therefore, be desirable to minimize their number.

In general, when the system is behaving well, fewer probes are required. For example, placing probes in the dispatchers that report performance is enough to detect that all requests are being served adequately, if performance is the only quality attribute of interest. However, when a problem is detected it may be desirable to have more detailed information to do diagnosis. This suggests that a useful capability would be to be able to add or remove probes dynamically depending on the tradeoff between diagnosis accuracy and probing impact.

? Research Questions

RQ 7 How to decide whether to dynamically adjust probe deployment, maintaining a balance between probing cost and diagnosis accuracy?

Chapter 3

Approach

In this section, we summarize our approach to autonomic fault detection and diagnosis, explaining how we address the research questions enumerated in Section 2. A key idea supporting our approach is to map observable low-level events in a running system to high-level models for which correctness specifications can be provided, as described in Sections 3.1.1 and 3.2. The high-level models used are software architecture models, as explained in Section 3.1. The identified architectural behaviors are then classified (Section 3.3) and failures mapped to software architecture components (Section 3.5). Analysis of the behavior models, as described in Section 3.6, helps system designers identify potential errors in the behavior specifications and improve their quality.

3.1 An architectural approach

The centerpiece of our approach is to use architectural models (specifically, component and connector views [22]) as a basis for fault detection and localization. We do this for several reasons. First, because architectural models provide a high-level perspective on the system, they simplify the reasoning process about system structure, behavior and repair. A system like the one presented in Chapter 2 performs computations concurrently on a large set of data and its components interact using a variety of low-level communication protocols. However, the *web server*, and HTTP protocol are natural abstractions for web developers. It comes as no surprise, therefore, that description of erroneous behavior in web applications is commonly described in terms of these abstractions. For example, failures described in "Causes of Failure in Web Applications" [60] refer to "error notices such as 404" and "blank web pages".

Second, architectural models scale to large and complex systems. Architectural models can express such systems with simple models through the use of abstraction thereby enhancing the scalability of our techniques. Very large, complex systems such as the Google File System [34] can be represented with a small number of components and connectors types. These models not only simplify reasoning about the system, but also reduce the search space for sources of the failures. They also present diagnosis at the same level of abstraction as architecture-based repair mechanisms such as Rainbow [33], simplifying the integration of diagnosis with the rest of the self-adaptive loop.

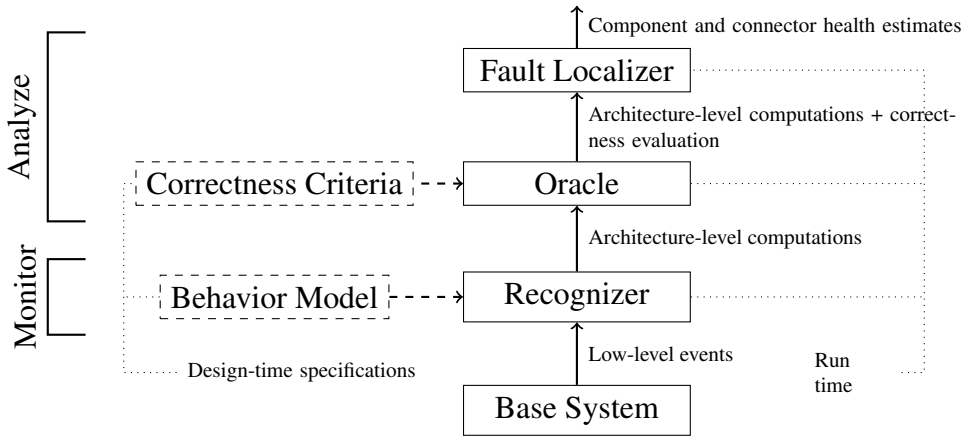


Figure 3.1: Overview of the approach.

Third, architectural models support the definition of systemic properties such as availability and performance [9], key types of properties we would like to use for diagnosis. Thus, by using architectural models, we can detect systemic failures and pinpoint the high-level components and connectors causing them. For example, a performance degradation in the response time noticed by clients in the example of figure 2.1 may be caused by a slow database.

Fourth, the design of architectural models often follow common patterns [12, 22]. By coupling our approach to architectural models, we can leverage recurring architectural forms to enhance reuse of behavior specifications across similar systems.

3.1.1 Diagnosis system overview

While software architecture can potentially provide a basis for addressing many of the challenges of diagnosis for autonomic systems, architectural-level computations are not directly observable in practice since only low-level events can generally be observed. For example, in the example web system presented in Chapter 2, we would like to reason about HTTP requests. But these requests may not be observable in practice: we can probe systems using a variety of off-the-shelf techniques such as aspects, standard network-based monitoring tools, or wrappers around system calls, but all of these will report low-level events such as POSIX system calls (e.g., `read(2)` and `write(2)`). This means that some recognition mechanism must translate low-level events of interest to high-level, architectural computations. Once we have carried out that abstraction step, we can then evaluate architecture-level behavior to determine the existence and location of faults.

Our overall approach to abstracting over low-level observations and checking them is shown in Figure 3.1. First, we feed the observed low-level events into a *Recognizer* that identifies architecture-level computations by matching the low-level events against a *Behavior model*. This model specifies patterns of events and how they are related to elements in the architecture.

The high-level computations are then analyzed by an *Oracle* to check whether they represent correct or incorrect behaviors, according to some *Correctness Criteria*. Correctness is derived

from the business rules that define the system's functional requirements and quality attributes. For example, an incorrect computation can occur because a protocol did not complete successfully, or it completed, but without satisfying desired performance requirements.

Recognized computations are then passed to the *Fault Localizer*, which periodically analyzes a subset of them using a fault localization algorithm to determine the health of each element in the computation. The *Fault Localizer* then reports the health of each element involved in the computation, which can be used by a self-adaptive system (such as Rainbow) to plan adaptations to repair unhealthy elements (outside of the scope of this thesis).

We now discuss each of the parts of the system in more detail.

3.2 The recognizer and the behavior model

Identifying architecture-level computations directly from the observed low-level behavior relies on significant assumptions on the implementation and makes specifications difficult to write and almost impossible to reuse. For example, to detect that a server has established a connection to another we could monitor network packets. A change of communication protocol from TCP/IP to UDP/IP would require new specifications, even if the change occurs only at a networking level. And a change in probes to monitor the servers instead of the network would also require new specifications.

These specifications are not reusable because they rely on assumptions on the whole implementation stack. At a high level, we care that one server to connects to the other, but we have also bound that behavior to the actual implementation mechanism (a network connection) and the actual probing system (reading the network).

This makes specifications highly specific and non-portable across systems. This also makes specifications complex as it requires writing rules to identify high-level behavior from patterns of probed low-level behavior.

To address this problem, we decompose the system behavior hierarchically, where detection of lower-level behavior feeds the detection of higher-level computations. Figure 3.2 illustrates a partial hierarchy of computations that could be applicable for the example in section 2. For example, the higher-level computations (dynamic web request and static web requests) can be computed from the proxied requests – HTTP requests sent to the load balancer and forwarded to a web server – and database queries.

Listing 3.1 presents specifications in the language used to define system behavior that will be introduced incrementally throughout this section and will be described in more detail in section 4.

In listing 3.1, four simplified types of computations (each declared as a `computation type`) are defined: `pxr`, a proxied request, `dbq`, a database query issued from a web server, `dwr`, a dynamic web requests and `swr`, a static web request. All four computations inherit properties from the generic `htc` (host/thread computation) in "package" `tc` (threaded computation). The threaded computation package defines the concept of a thread running in a host.

```
1 import "tc.call"
2
3 // Defining of HTTP result codes.
4 typedef int32 http_result {
```

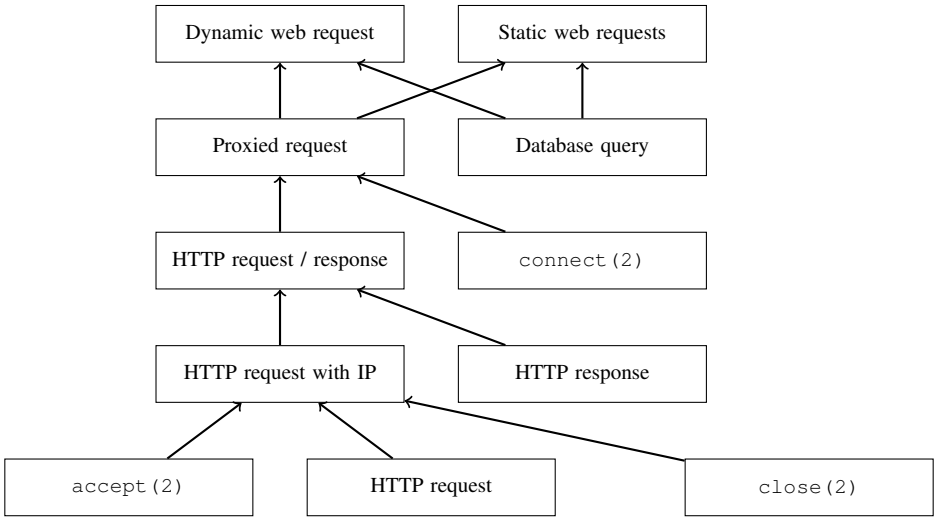


Figure 3.2: Partial hierarchy of computations for the example in section 2 including the low-level events that are system calls like `accept(2)`.

```
5       invariant self >= 0 and self <= 599;
6   }
7
8   // Proxied request: send from client to dispatcher and
9   // then to a web server (and then back to the client).
10  computation type pxr : tc::htc{
11      result : http_result;
12  }
13
14  // Database query.
15  computation type dbq : tc::htc{}
16
17  // Static and dynamic web requests.
18  computation type dwr : tc::htc{
19      result : http_result;
20  }
21  computation type swr : tc::htc{
22      result : http_result;
23  }
```

Listing 3.1: Simplified definition of dynamic web requests.

The language contains primitives akin to object-oriented languages. This similarity has the advantage of easing the learning curve for software engineers and bringing the power of established development techniques. For example, *computation types* are akin to *classes* and *families* to *namespaces* (or *packages*).

Recognizing higher-level computations from other computations is done through the definition of *recognizers* as illustrated, in simplified form, in Listing 3.2. Static and dynamic web requests are, essentially, equal, except that dynamic web requests involve a database query, whereas static web requests do not. In our recognition language, we can now express the difference in the two computations.

```
1   // The dwr recognizer
2   recognizer dwr_rgn(r : pxr) {
3     invariant exists q : dbq | r->during_same_thread(q);
4     emit dwr(result: r.result);
```

```
5 }
6
7 // The swr recognizer
8 recognizer swr_rgn(r : pxr) {
9   invariant not (exists q : dbq | r->during_same_thread(q));
10  emit swr(result: r.result);
11 }
```

Listing 3.2: Identifying dynamic web requests and static web requests.

The two recognizers in Listing 3.2 identify the static and dynamic web requests from proxied requests depending on whether a database query is made during the process of the request or not. The `invariant` clause contains the logic condition under which the computation is recognized. The `emit` clause defines which computation is identified and how its contents are derived from the lower-level computations.

In plain English, a proxied request is a dynamic web request when *there is at least one database query performed during the request by the same thread*. The code in listing 3.2 contains an *invocation* of a *method* named `during_same_thread`. Invoking a method is akin to the object-oriented concept with the same name, which, loosely speaking, means executing an algorithm. This method is invoked in a computation of type `pxr` but is actually defined in its super type, the `tc::htc` computation type. This allows reuse of the definition for all subclasses of `tc::htc`. This method is defined as in Listing 3.3.

```
1  family tc {
2      computation type htc {
3          // ...
4          bool during_same_thread(c : htc) {
5              return same_thread(c)
6                  and start(c) >= start(this)
7                  and end(c) <= end(this);
8          }
9          // ...
10     }
11 }
```

Listing 3.3: Definition of the DURING_SAME_THREAD method.

3.3 The oracle and correctness criteria

Once architectural-level computations have been identified, it is necessary to determine which of those computations refer to correct and incorrect computations. This is done using an *oracle*, which performs *fault identification* (but not *fault localization*). It bases its decision on *correctness criteria*. We use a loose definition of correctness criteria that allows not only functional correctness, but also checking other properties like latency.

The oracle receives information about high-level computations performed by the system and, using the correctness criteria, marks them as failures or successes. Correctness criteria are defined as predicates over computation types. An oracle can be seen as a "failure detector." If multiple oracles are defined for the same computation, a computation is considered a failure if any of them marks it as a failure.

Marking a computation as a *failure* in an oracle has no effect on recognizers. Failed computations can still be used to recognize other computations (even if they arrive later). And it also

does not, by itself, affect the correctness of computations *based on* that computation: recognizers and oracles work independently, and every computation is evaluated independently by applicable oracles.

This decoupling of the recognizer and the oracle allows detection of failures that have been masked by recovery mechanisms: for example, if a web server fails and the load balancer redirects the request to another web server, the client will perceive correct behavior and the high-level computation will succeed. An oracle could still flag the intermediate failure to allow localizing the source of the masked failure.

To ease reuse, the oracle is split into multiple *oracle instances*, each one classifying a subset of all computation types known to the system. These oracle instances are also typed and their types can be sub-classed to form hierarchies and further enhance reuse in a typical object-oriented way.

```
1 oracle request_latency {
2     bool evaluate(cr : dwr) {
3         return end(cr) - start(cr) < 2s;
4     }
5 }
```

Listing 3.4: Example oracle that evaluates the latency of a request.

Listing 3.4 contains an example of an oracle that identifies failures when the time between a request and a response exceeds 2 seconds. The `evaluate` method is invoked on computations of type `dwr` (dynamic web request) and return `true` when the time between the begin of the call/return and the end of the call/return is less than 2 seconds.

3.4 Keeping track of architectural elements

An oracle, as described above, outputs computations with a boolean classification of success or failure. To perform localization it is necessary to be able to associate these computations with the architectural elements that contributed to each computation. This is done by associating probes with architectural elements and keeping track of which probes generated which computations.

For example, in Figure 3.3 we have a four-component system where three low-level computations, x, y and z have been detected in the three components, Componnet 1, Component 2 and Component 3, respectively. Computation A is recognized from x and y and will be associated with both Component 1 and Component 2. Computation B is recognized from A and z and will be associated with all three components.

In CALL probes are associated with architectural elements, both components and connectors. For example, probes for the example in Figure 3.3 could be defined as in Listing 3.5.

```
1 // Define probe_x that reports events of type x located at component_1.
2 probe probe_x : x @ component_1;
3
4 // Same for remaining probes.
5 probe probe_y : y @ component_2;
6 probe probe_z : z @ component_3;
```

Listing 3.5: Example probes for example in Figure 3.3

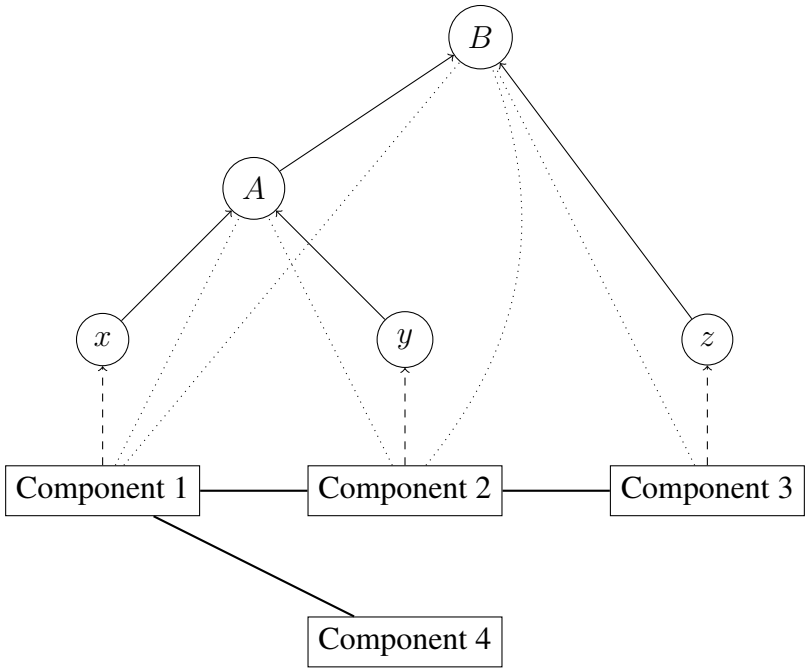


Figure 3.3: Keeping track of the architectural elements associated with computations.

The architectural elements are an input to the framework. No method to identify the architecture or to keep track of architectural changes is proposed. The architectural elements for architectures that don't change at runtime are typically defined in the CALL specification, like in Listing 3.5.

If using the runtime, CALL-RE, with a full MAPE loop strategy execution engine like Rainbow [33], or if discovering the architecture at runtime using a framework such as DiscoTect [79], providing the architecture in the CALL specification is not possible. In that case, the Java API provided by CALL-RE can be used to dynamically update probes and recognizers.

New recognizers and oracles can also be added at run time, which might be necessary if new elements are added to the system. Some limitations apply, however. Recognizers and oracles can be added, but only as new instances of existing types, as described in see Section 4.1.2.

3.5 The fault localizer

The outputs provided by oracles, matched to architecture elements, *i.e.*, components and connectors, in the system, is finally fed to the fault localizer, the last element in the chain presented in Figure 3.1.

The output of oracles contain a list of architectural elements, *.i.e.*, components and connectors, and an evaluation of correctness. This data structure is akin to the data that is collected by instrumenting systems for spectrum-based fault localization at design time [3, 42, 78].

Spectrum-based fault localization algorithms were designed to localize faults by running tests at design time. Humans write test code and correctness criteria. While running, the test collects instrumentation on which parts of the code (classes, methods, code blocks) were used for execu-

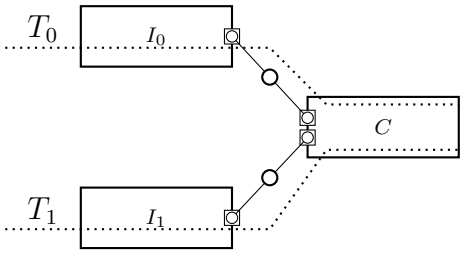


Figure 3.4: Example system with 2 input functions, a computation function and two traces.

I_0	I_1	C	Correct?
0	1	1	Yes
(1)	0	(1)	No
0	1	1	Yes
(1)	0	(1)	No
0	1	1	Yes
1	0	1	Yes
0	1	1	Yes
(1)	0	(1)	No
0	1	1	Yes
(1)	0	(1)	No

Table 3.1: Example of spectrums in fault localization at design time.

tion, *i.e.*, a *trace*. The list of traces and correctness evaluation is then used to determine which elements are faulty.

For example, consider the executions of a program with two input functions, I_0 and I_1 and a computation function C as depicted in Figure 3.4. Suppose we have two traces, T_0 and T_1 as depicted. If trace T_0 represents an incorrect trace and T_1 a correct trace, where is the fault located? Intuitively, we can guess that I_0 is more likely to have caused the failure since, if C was faulty, T_1 might have also failed. However, with just two traces the confidence in such an evaluation is low, especially considering the possibility of intermittent faults. It is also likely that C is intermittently faulty and the failure happened to occur with I_0.

If a larger set of traces is collected the evaluation becomes clearer. For example, consider the trace of ten hypothetical tests in Table 3.1.

There are four faults in ten runs with all faults containing component I_0 and component C. Either I_0 could explain the fault or C could. However, the evaluation is quite different when compared to the case with just two traces. We can assume I_0 has a failure probability of 80% or we can assume C has a failure probability of 40%. However, in the latter case, *all* of the failures with C would have happened with I_0. With a total of 10 traces this is unlikely, but still realistic. As the set of traces increases if all the failed traces continue to only include I_0 and C, it becomes less and less likely for C to be a good explanation for the failures.

Design-time fault localization algorithms look at the list of exercised areas of the code along

with the test result (called a *spectrum*), similarly to what was done above, and compute the set of likely locations for faults typically using a probabilistic model. This result is commonly a probability distribution: each set of components is assigned the probability that it is responsible for the faults identified.

Different spectrum-based fault localization algorithms will produce different outcomes based on different models. But typically they will produce a probabilistic output of how likely individual components are to be faulty or how likely a set of faulty components is responsible for the observed failures [3, 75]. As was shown in the example above, the more traces observed, the more confidence there is in the output and, for software systems with large numbers of automated unit tests, the results are quite accurate. With very few traces, or for components exercised very rarely, the confidence will in general be low and, in some cases, it is impossible to distinguish individual components [19].

At run time, we don't have test runs, but we have high-level computations. We don't have human-written test evaluation code, but we have human-written oracles. We don't have instrumented code, but we have the set of architectural elements that were used to produce each computation.

The similarity of the data structures for both design-time and run-time scenarios suggests that these well established design-time algorithms could be used at run time. One important difference between both scenarios is that in the design-time execution case we wait for all tests to run and collect all the traces before performing a diagnosis. But at run time we cannot wait for all the traces to produce a diagnosis.

Not being able to collect all traces means that we need some way to decide when to run the algorithm and which traces to run the algorithm with. To make a decision that supports the goals of this thesis, we need to consider three factors. First, the frequency of algorithm execution determines how quickly after a failure can we determine the source of the failure. Second, the further back we look in time when accumulating traces, the more information we will have for diagnosis and the more likely it is that we can correctly pinpoint the location of the failure. Third, design-time fault localization algorithms were designed assuming that the set of faults is fixed and if they run over sets of traces that cover a period in which the set of failures has changed, they are more likely to misidentify the current set of failures.

Let's take a more detailed look at the third factor. At run time, the actual set of failures can change (hopefully not too often). For example, a server may experience a hardware fault. This means a set of traces may cover a period containing traces with failures, which increase the confidence the failed components are failing, but will also contain traces before the failure occurred, that increase the confidence those components are healthy. A failure diagnosis from such a mixed set of traces is likely to produce less confident outcomes than a diagnosis from a set of traces that covers a period in which there are no changes in the set of failures.

The third factor is at odds with the second factor: the more traces we feed the algorithm, the more accurate the output. In fact, if we analyze too few traces, we may not be able to distinguish failures at all (as we have shown in [19]).

This conflict raises a challenge: the larger the period over which we collect traces, the more accurate the diagnosis, except if there has been a change in the failed components, in which case larger collection periods will make the diagnosis less accurate. And there is no way to detect when the changes in the set of failed components happens as that is exactly the problem we're

trying to solve.

We propose a solution that is based in three principles: (1) the algorithm runs regularly (addressing the first factor), (2) between each run we collect a set of traces that feed the next run of the algorithm, and (3) when the algorithm runs, if the output is inconclusive, the traces that were used are kept to the next run. In our solution the algorithm is run at a high frequency and the set of traces fed to it is small: most algorithms are very fast on small data sets and they are quick to conclude if everything is OK, *i.e.* there are no failures on the system. If there is a failure, the algorithm is run with a small set of traces, which may be inconclusive. If so, we collect more data before the next run and we keep increasing collection until we have gathered enough data.

We use a *time window* of size Δ that is the period at which the algorithm runs. This time window increases by Δ if there is an inconclusive diagnosis. When a conclusive diagnosis is produced, the time window is reset to Δ.

But when do we know whether a diagnosis is conclusive? We assume algorithms output probabilistic distributions over the components (most do). With an assumption that the output is a probabilistic distribution, we decided to use *entropy* to evaluate conclusiveness of diagnosis. *Entropy* (borrowed from Information Theory [70]), also known as Shannon Entropy, characterizes the (im)purity of an arbitrary collection of, in our case, diagnosis candidates. Because entropy measures the average unpredictability of a random variable, it serves as a rather direct measure of diagnosis accuracy. The idea is to adapt the time window given the entropy, knowing that more information decreases the entropy of the ranking.

Therefore, depending on the rate at which computations are generated and the information associated with them, this window will change as the system runs. Once a set of computations have been collected in a time window, they will be fed into a design-time fault localization algorithm.

How does this work in practice? Our implementation defines two parameters: a time rate parameter Δ and a maximum entropy H_m. We perform diagnosis every Δ and our time window, W, is set initially to Δ.

At regular intervals of Δ we apply fault localization to all computations that completed in W. The algorithm produces a set of candidates d_k ranked probabilistically ($Pr(d_k)$) and we compute the entropy of the distribution, $\mathrm{H}(D)$.

If $\mathrm{H}(D) \leqslant H_m$ then we consider the diagnosis to be accurate and we output the result of the diagnosis. If $\mathrm{H}(D) > H_m$ then we need to collect more data and increase W by Δ. This means that the next time we apply the algorithm, we will use all the data we have plus all computations that finished in the last Δ.

We reset the time window to Δ as soon as we produce a diagnosis result to start collecting data for the next diagnosis. Because W increases with Δ and we compute the diagnosis every Δ, past information is prevented from interfering in future diagnosis.

If we consider the full MAPE loop and not just monitoring and analysis, it is possible that both setting Δ and deciding when to reset could be part of later stages of the loop. For example, an execution engine may want to reset the traces upon execution of a strategy such as a reboot. See Section 9.4.2 for a discussion on future work on integration with the full MAPE loop.

The framework provides a general placeholder for an algorithm that outputs a map of components to their fault probabilities given a set of architectural computations classified as successes or failures. It can operate with any of the multiple algorithms that fit that contract. A default

implementation is provided based on Barinel [3].

3.6 Design-time analysis of specifications

In the realized framework, humans write the specifications for the recognizers and oracles. This is a critical activity but, like all programming and specification activities, it is error-prone. Analysis helps to improve quality by identifying probable mistakes and ensuring desired properties.

There are three analyses that we perform on the behavior specifications and correctness criteria written in the framework's declarative language: consistency analysis (Section 3.6.1), accuracy analysis (Section 3.6.2) and finiteness analysis (Section 3.6.3).

3.6.1 Consistency analysis

The purpose of consistency analysis is to detect specifications that contain the equivalent of "dead code" in programming languages. It detects (1) recognizers that are never used, (2) recognizers whose output is not useful, and (3) oracles that are never used.

Consistency analysis ensures that (1) all recognizers could be used in at least one system execution, (2) the output of every recognizer is used either for another recognizer or for an oracle in at least one system execution, and (3) every oracle will be used in at least one system execution. Consistency analysis works by building a graph of all probes, the *computation graph*. This graph links all probes, with information about the architectural elements they are associated with, to all recognizers and oracles.

To illustrate consistency analysis, consider the example in Figure 3.3. Assume, in addition to what's represented in Figure 3.3, that a probe on component 4 identifies computations of type `w` and a recognizer can identify computations of type `C` from `x` and `w`. The computation graph would be the one depicted in Figure 3.5. In this computation graph, the probes for computations `x` and `y` (named `px` and `py`, respectively) will feed their computations to the recognizer for computation `A` (named `RA`). Probe `px` will also feed its computations to the recognizer of computation type `C` (named `RC`). Probe `pz`, identifying computations of type `z` on component 3, will feed its computations to recognizer `RB` that recognizes computations of type `B`. Computations of type `B` are evaluated by oracle `O0` and computations of type `C` are evaluated by oracle `O1`.

This graph allows identifying probes and recognizers whose output is not used, recognizers with missing inputs and oracles with no inputs by looking at the graph and looking for missing edges.

Because we don't analyze recognizer invariants for contradictions (recognizers' invariants are written using predicate logic as described in Section 3.2), consistency analysis is approximate as we assume all recognizers' invariants can potentially evaluate to true. Consequently, while consistency analysis will not generate false positives, meaning it will never mark reachable recognizers or oracles as unreachable, it may generate false negatives, meaning it may report that all recognizers and oracles are reachable when they are not. This is akin to dead code analysis in which code marked as dead code is dead code, but some dead code may not be marked.

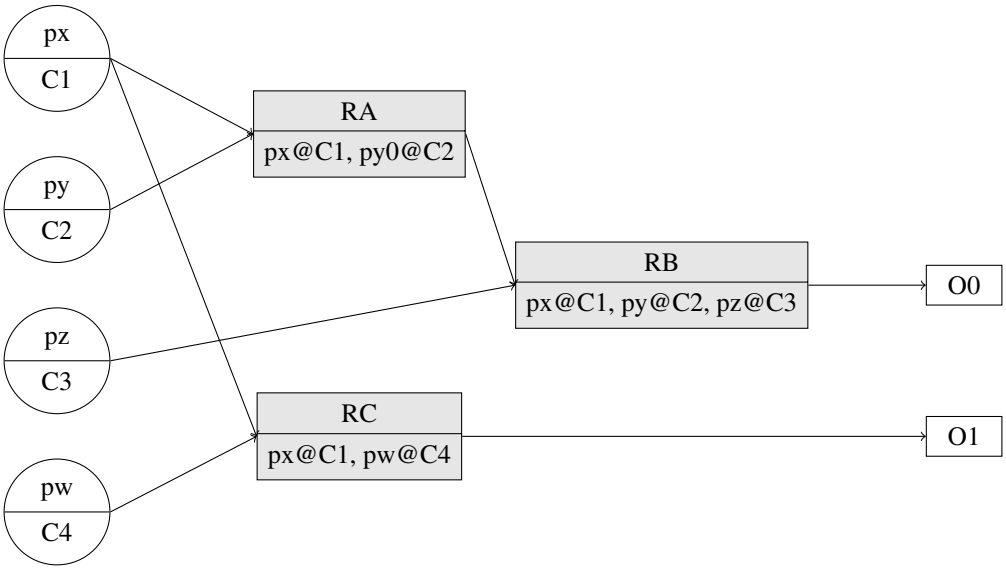


Figure 3.5: Computation graph for the example in Figure 3.3.

Consistency analysis is important because of the language's declare-and-wire approach, which will be described in more detail in Chapter 4. In the realized framework, recognizers and oracles are declared and then they are "wired" declaratively. This analysis helps ensure that this wiring activity does not leave recognizers or oracles without inputs or leave probes or recognizers without their outputs connected.

3.6.2 Accuracy analysis

The accuracy analysis will, for a given behavior model, correctness criteria and probe placement, identify what components cannot be individually identified if one of them fails. For example, if two components are always used together, it is not possible to distinguish between a failure in one of the components from a failure in the other.

Diagnosis accuracy is a partition of system's components in which two components belong to the same partition if we cannot distinguish, with the current set of recognizers and oracles, which of them failed. As with consistency analysis, and for the same reason, accuracy analysis is approximate: it may report components as being individually identifiable when they are not, but it won't report the opposite. In other words, the analysis is sound, but not complete.

Accuracy analysis is done by building the *full trace matrix*. The full trace matrix is obtained by generating all possible traces from probes to oracles in the computation graph. For example, looking at the example in Figure 3.5, it is possible to identify two traces: one leading up to oracle `O0` that involves components `C1`, `C2` and `C3` and another leading up to oracle `O1` that involves components `C1` and `C4`. The full trace matrix is in Table 3.2.

In this matrix it is possible to see components `C2` and `C3` will always be used together. This means they are not distinguishable by the fault localizer. The fault localizer will provide the fault probability of both `C2` and `C3` *combined*, but won't provide individual component estimates.

C1	C2	C3	C4
px	py	pz	-
px	-	-	pw

Table 3.2: Full trace matrix for the example in Figure 3.5.

It is also possible to see in Table 3.2 that the traces are still distinct if we remove px and either py or pz. This allows the planning stage of the MAPE loop to balance the cost of deploying probes px, py and pz. This is further discussed as future work in Section 9.4.2.

If we remove all three probes py, pz and pw, then both traces are still *observable*, but no longer *distinguishable* [19]. This is not particularly useful for diagnosis in isolation, but when coupled with the MAPE loop, can be used to cheaply probe if there are failures and then deploy probes to allow localization. This technique of dynamic probe placement may be advantageous in some scenarios where the cost of diagnosis delay outweights the cost of probe execution.

3.6.3 Finiteness analysis

Finiteness analysis will detect whether any recognizer may need to store an unbounded number of computations. This may happen if a recognizer needs to keep computations for an undetermined amount of time. For example, in the system described in section 2, if we are allowed unbounded time for a server to respond to a request, the recognizer may need to keep track of an unbounded number of requests waiting for a response.

The finiteness analysis is also an approximate analysis, as some recognizers may be practically bounded while appearing formally unbounded. This is a consequence of one of two factors. First, the analysis is approximate and may generate false positives – signaling a recognizer as unbounded when it is not. See Section 5.2.6 for details on how the prediction of future events works in discarding past events from recognizer's input.

Second, there may exist external constraints that are not specified in the formula. For example, a recognizer that matches file opens with file closes will need to keep computations that represent file open operations in memory until the event that closes the file is detected. Such a recognizer may seem unbounded as a program could, theoretically, open an unbounded number of files. However, in practice, there are only so many file descriptors that can be kept open by a program and so such a warning can be safely ignored by a system designer.

3.7 General properties of the framework

In this research we show that the framework possesses several general properties. Specifically, diagnosis performed by the framework is:

Complete: Diagnosis will not generate false negatives with respect to the behavior definition. Probed traces that are invalid according to the behavior definition will be flagged as such. Recognized computations that are invalid according to the correctness criteria will also be flagged as such.

This is guaranteed by building recognizers and oracles based on predicate logic guaranteeing deterministic and accurate evaluation of computation patterns and accurate computation classification.

Correct: Diagnosis will not generate any false positives with respect to the behavior definition. Computations that are marked as a failure are violations of the specified behavior.

Just like completeness, this is guaranteed by the use of predicate logic in recognizers and oracles.

Consistent: Computations will eventually be marked as either correct or incorrect and never both. The diagnosis system is *decidable*.

While arbitrary predicates can be used in both recognizers and oracles, these predicates cannot form or contain loops or any type of recursive iteration, thus guaranteeing decidability.

Monotonic: Given two sets of observations of the system in the same health state, one being a strict superset of the other, the diagnosis of the larger set will always be reported with the same or higher probability of being the correct one.

This is a guarantee from the default fault localization algorithm used, Barinel [3]. Using a different fault localization algorithm may violate this property if the algorithm itself does not satisfy this property.

Chapter 4

The CALL language

In Chapter 3 we gave an overview of our approach to fault detection and diagnosis. This approach relies on defining computations, using recognizers to abstract lower-level computations into higher-level computations, using oracles to classify computations and using a fault localizer to pinpoint the most likely causes of failures.

In this section we present the language, named CALL – Computation Abstraction and Localization Language – used to instantiate the architecture presented in Chapter 3. We describe the language in detail, providing rationale for its design and features, describing its concepts and providing an introduction to its syntax (the full syntax is described in Appendix B).

This chapter is organized as follows: Section 4.1 describes the key decisions made in CALL's design to support both the required functionality and quality attributes. Section 4.2, the CALL language is described informally, exploring its core concepts and features.

4.1 Key Design Decisions in CALL

Functionally, CALL was designed to support the architecture in Figure 3.1. CALL is, in essence, a language that supports processing of computations to abstract lower-level computations into higher-level computations (performed by recognizers), classification of computations (performed by oracles) and mapping of computations to the locations in the architecture where they were preformed (to support the fault localizer).

In addition to supporting the framework's architecture, these language design decisions also support some of the required quality attributes, described in Chapter 1, which manifest themselves as cross-cutting concerns: (1) analyzability to prevent human error and ensure, to a limited degree, correct behavior, (2) similarity to existing programming languages to lower barriers to use, and (3) reusability to lower the cost of instantiating the framework for a given system.

Section 4.1.1 describes the design decisions that support the functionality required and Section 4.1.2 describes how the quality attributes are supported.

4.1.1 Functionality

In brief, the key design decisions in CALL are:

- DD1: Computations are first-class entities in the language.
- DD2: Recognizers are first-class entities that use patterns to identify high-level computations from streams of lower-level computations.
- DD3: Oracles are first-class entities that use predicate logic to classify computations.
- DD4: The target system's architecture is described in CALL and computations are mapped into the target system's architecture.

DD1: Computations are first-class entities in the language

The concept of a computation was introduced in previous chapters. It was briefly described in Chapter 3, where we illustrated how it can be used to detect failures in the target system. CALL needs to describe computations, match computations, produce new computations and classify computations. Computations are a fundamental concept in CALL and therefore, they are first-class entities.

Being a first-class entity means computations are explicit constructs of the language, they are typed and they can be bundled into libraries for reuse. To support reuse, computation types support inheritance, allowing common functionality to be abstracted away into super-classes, a common feature of object-oriented languages.

Explicitly typing computations allows analyzability of the program supporting the detection of missing probes or incorrect wiring of recognizers and oracles. Inheritance allows reusing recognizers and oracles defined for more general types. Section 4.2.2 shows how to define an abstract recognizer based on abstract computations for the call/return architectural style and how it can be reused to define a specific computation.

Computation types are declared using syntax that is similar to the declaration of structures (records) used in mainstream programming languages. For example, consider the following CALL code:

```
1 computation type http_request {
2     request_type : string;
3     requested_path : string;
4 }
```

Listing 4.1: Example HTTP request computation type.

This declaration is similar to its C++ equivalent, sharing some of its semantics:

```
1 struct http_request {
2     std::string request_type;
3     std::string requested_path;
4 };
```

Listing 4.2: Equivalent example in C++.

A software developer familiar with C++ (or any language with records) will understand that there is a type `http_request` and that individual computations are instances of this type.

While computations have some similarities with structures in C++, they also have some unique features that will be described in detail later in this chapter:

- They have a start and end time that are automatically maintained by CALL.
- They are associated with elements in the system's architecture.

DD2: Recognizers are first-class entities that use patterns to identify high-level computations from streams of lower-level computations

Figure 3.1 shows that computations are detected in the target system by probes and flow into recognizers. These flows of computations are named *streams* (see Section 4.2.4 for a discussion of streams). Recognizers look at these streams to match patterns of computations. Each match yields a high-level computation.

The specification of the patterns uses predicate logic. Predicate logic was chosen because (1) it is a well-known declarative way to express relations between entities, (2) it is expressive enough to cover most use cases, and (3) it provides compile-time analyzability of the patterns (see Section 3.6.3). This last benefit is critical to CALL: it enables dealing with uncertainty about future computations (see Section 5.2.1 for a discussion on time windows and on dealing with future uncertainty).

Recognizers are one of the fundamental concepts in CALL and, therefore, are first-class entities in the language: they are explicit constructs, they are typed and they can be stored in libraries for reuse.

The logical formulas are used to find patterns of lower-level computations. When they evaluate to true, a higher-level computation is identified (or *recognized*). These formulas are known as *recognizer invariants*. In simple cases these formulas reduce to elementary comparisons. For example, a recognizer that matches a computation of type `A` with a computation of type `B`, provided they have the same value in some property `c`, could be expressed as `a.c == b.c`:

```
1 recognizer example_recognizer(a : A, b : B) {
2   invariant { a.c == b.c }
3   emit D(c: a.c);
4 }
```

Listing 4.3: Example recognizer.

This recognizer matches computations of type `A` with computations of type `B` providing a higher-level computation of type `D`. It matches all `A`s with `B`s as long as they have the same `c` property and the resulting `D` computation will have property `c` initialized with the same value as `a`.

While simple predicates are expected to be common in CALL, many recognizer patterns require the use of quantifiers. For example, the following recognizer identifies computations of type `A` for which there are no computations of type `B` that match property `c`:

```
1 recognizer example_recognizer(a : A, &b : B) {
2   invariant { not (exists bb : &b @ bb.c = a.c) }
3   emit D(c: a.c);
4 }
```

Listing 4.4: Example recognizer.

Here, the `&b` construct refers to a *stream* of computations whose type is `B`.

One final but important note on the use of quantifiers concerns decidability. Although satisfiability of general predicate logic is undecidable, recognizer evaluation is decidable even though it relies on quantifiers. The reason is that at any point in time, the number of computations in each stream is finite and all quantifiers in the recognizer invariant can be expanded into *and*s or *or*s. More details are available when discussing the formal semantics of the language in Chapter 5.

DD3: Oracles are first-class entities that use predicate logic to classify computations

Oracles are, like recognizers, first-class entities: they are explicit constructs of the language, they are typed and they can be bundled into libraries for reuse.

Oracles classify computations using quantifier-free predicate logic. Unlike recognizers that look for patterns of computations, oracles only look at a single computation to decide whether a computation represents a failure or not and, therefore, have nothing to quantify over. Consider the following example:

```
1 oracle type c_is_positive {
2     bool evaluate(d : D) {
3         return d.c > 0;
4     }
5 }
```

Listing 4.5: Example oracle.

Here, the oracle `c_is_positive` classifies computations of type D as being "correct" or "incorrect". It marks as correct those whose property c is positive and incorrect all others. The correctness criteria is *declared*: we specify only the relation between the elements in the computation.

DD4: The target system's architecture is described in CALL and computations are mapped into the target system's architecture

Mapping failures to faults (see Appending A) requires *localizing* the faults – that is, identifying which parts of the system have failed. Since the framework maps failures to faults at the architectural level (see Chapter 3), CALL needs to contain some form of architectural description.

CALL's architecture description is based on Acme's [32] concepts. It supports typed components, connectors, ports and roles and typed attributes in all these elements. It also supports, like Acme, organizing these types in families for reuse.

Acme can represent a given architecture, but has no constructs to allow dynamic updates to the architecture. CALL was designed to be part of a MAPE loop and the system's structure is expected to change at least as a result of failure identification. This means dynamic updates are a required feature in CALL. A software engineer familiar with Acme (or other similar architecture description languages) should find the concepts familiar, with the exception that the language supports dynamic constructs. For example, the following snippet creates an architectural description of a simple pipe-and-filter system with a source and a sink:

```
1  architecture cmodel {
2      component source {
3          port source_out;
4      }
5      component sink {
6          port sink_in;
7      }
8
9      connector pipe {
10         role pipe_in;
11         role pipe_out;
12     }
13 }
14
```

```
15 bind source.source_out to pipe.pipe_in;
16 bind sink.sink_in to pipe.pipe_out;
```

Listing 4.6: Trivial pipe and filter described in CALL.

CALL maintains an architectural model of the system at run time, and the model is updated dynamically, programatically, as the CALL-RE receives notifications from external systems. See Section 3.4 for a description of how architectural elements are used in CALL.

4.1.2 CALL Quality Attributes

The quality attributes that the language design addresses are:

- **Analyzability to prevent human error and ensure, to a limited degree, that incorrect specifications are identified**. Analyzability is achieved by using predicate logic to establish relations in recognizers and to define evaluation criteria in oracles. Predicate logic is amenable to automated analysis and is decidable and compact, making the formulas less error-prone.

  For example, consider a computation type `C` that has two properties: an ID named `id` and another set of computations of type `C` named `inner`. Checking that there are no computations in the inner set whose ID matches the outer computation's can be written using predicate logic (in CALL syntax) as:

```
1 invariant { #{ cc : c.inner | cc.id = c.id } = 0 }
2 // Alternatively
3 invariant { not (exists cc : c.inner @ cc.id = c.id) }
```

  An equivalent C++ implementation would be:

```
1 bool check(computation c) {
2     for (auto it = c.inner.begin(); it != c.inner.end(); it++) {
3         if (it->id == c.id) {
4             return false;
5         }
6     }
7
8     return true;
9 }
```

- **Similarity to existing programming languages to lower barriers to use**. While CALL introduces a new approach to describing and evaluating behavior, namely by declaratively specifying relations instead of requiring users to provide algorithms, CALL was designed to reduce the number of new concepts software engineers would have to learn. Additionally, making the syntax similar to other languages helps improve readability of the language.

  Consequently, the syntax for CALL is similar to:

  - C++ and Java for data types and logic statements and control structures.
  - SQL and Z for set-related predicates.
  - Acme for architecture-related concepts.

As an example of similarity to C++, consider the following definition of a computation type in CALL and a similar class in C++:

```
// CALL
namespace my_package {

computation type my_type : super_type {
    a : int32;
    b : string;

    int32 increase_a(int32 amt) {
        a = a + amt;
        return a;
    }
}

}
```

```
// C++
namespace my_package {

class my_type : super_type {
    int32_t a;
    std::string b;

    int32_t increase_a(int32_t amt) {
        a = a + amt;
        return a;
    }
};

}
```

Invariants, while not easily mapped to C++ or Java, are similar to SQL syntax. This is because CALL operates on sets of computations (see Chapter 5 for details) similarly to how SQL operates on tables.

```
// CALL
not (exists cc : c.inner @ cc.id = c.id)
```

```
-- SQL
not exists (select * from c_inner where cc.id = c.id)
```

- **Reusability to lower the cost of instantiating the framework for a given system**. CALL allows abstracting computation types (and their data), recognizers and oracles. For example, a *threaded* library could define a `threaded` computation type and an abstract recognizer that matches two computations if they are in the same thread.

```
namespace threaded {

computation type threaded {
    thread_id : int32_t;
}

abstract recognizer type thread_match_r(t1 : threaded, t2 : threaded) {
    invariant { t1.thread_id = t2.thread_id }
}

}
```

The `thread_match_r` recognizer is *abstract* in the sense used by object-oriented languages: it cannot be used directly as it needs to be specialized.

The abstract types provided in libraries are specialized to particular systems. This promotes reuse and reduces the amount of CALL code that needs to be written for each individual system. The following example shows reuse of the `threaded` definitions above:

```
1  namespace reused {
2
3  computation type reused_type : threaded {
4      extra_data : string;
5  }
6
7  recognizer type reused_recognizer(r1 : reused_type, r2 : reused_type) {
8      recognizer threaded_r : thread_match_r;
9
10     invariant {
11         threaded_r(r1, r2) and r1.extra_data = r2.extra_data + "_extra"
12     }
13
14     emit new reused_type(threaded_id: r1.threaded_id, extra_data: r1.extra_data);
15 }
16
17 }
```

Use of multiple inheritance (described in more detail in Chapter 5), allows splitting types into multiple pieces that can be used together. For example, a computation can simultaneously be a `call_t` (from call/return), `threaded_t` (from threaded) and a `ip_t` (from TCP/IP).

4.2 CALL

Having described the approach in Chapter 3 and an overview of the key design decisions contributing to the design of CALL in Section 4.1, we now desribe CALL in more detail, using an example to illustrate the main language features. The language will be introduced progressively as we build the example program. This section is intended as an initial guide, or introduction, to the language. The full syntax is described in Appendix B.

4.2.1 Computations, Computation Types and Data Types

CALL processes *computations* that are detected by *probes* and recognized by *recognizers*. Computations in CALL are instances of types. Each computation type defines the properties that computations have. We have seen previously in this chapter and in Chapter 3 examples of computation types. In this section we dig into more detail on the concepts of computation, computation type and, more generally, data types, of which computation types are a specialization.

The example we'll use to illustrate CALL throughout this section is taken from the telecommunications world. We will be checking the functionality of a credit control system.[1] This system is used to check whether a subscriber on a pre-paid service has enough credit to access the internet. Every time the subscriber's device attempts to use its internet connection, the "router"

[1]The system described here is a simplification of a real production system the author has worked on.

will regularly check with a credit control system whether the subscriber is authorized and how much traffic is he authorized to use. This is done by sending credit control requests (CCRs) when the connection starts, while it is ongoing and when the connection terminates. The requests are sent to a credit control service that replies with a credit control answer (CCA). The requests will specify how much traffic has been used in the current session and the answer will let the "router" know how much it can allow. At the end, when the network session ends, a summary record (call data record, CDR) is generated and usually consumed by accounting software to display in invoices.

This system can be affected by several kinds of failures. For example, CCAs may be delayed or they may not be sent at all. Also, multiple CCAs can be sent as a response to a CCR. But failures on the "router" side are also possible: more data may be allowed than what was written in the CCA, for example.

We declare five computation types: `ccr_i_t`, (Credit Control Request – Initial) `ccr_u_t`, (Credit Control Request – Update) `ccr_t_t` (Credit Control Request – Terminate), `cca_t` (Credit Control Answer) and `cdr_t` (Call Data Record). The suffix `_t` is a convention for naming types.

All of these computation types require an IMSI (International Mobile Subscriber Identifier) – a string with a maximum of 15 characters. With the exception of the `ccr_i_t`, all of these computations also require a non-negative byte count.

These computation types can be defined in several ways. The simplest way is to define them directly with their properties:

```
1  namespace telecom {
2
3  computation type ccr_i_t {
4      imsi : string;
5  }
6
7  computation type ccr_u_t {
8      imsi : string;
9      usage : int32;
10 }
11
12 computation type ccr_t_t {
13     imsi : string;
14     usage : int32;
15 }
16
17 computation type cca_t {
18     imsi : string;
19     usage : int32;
20 }
21
22 computation type cdr_t {
23     imsi : string;
24     usage : int32;
25 }
26
27 }
```

This declaration suffers from two problems: it does not provide constraints for the values and it contains duplication of fields and types. In this case, given the simplicity of the declarations, the duplication is not problematic. But in more complex cases such duplication makes maintenance and reusability more difficult and error prone.

The lack of constraints is more problematic, as it allows a probe to report computations that are valid as far as CALL is concerned, but invalid in the business domain: for example, reporting an IMSI of 20 characters or a byte count of `-10`. Allowing these invalid values would force us to tell CALL what to do with those cases. This usually means introducing these conditions as part of the recognizer's invariant to avoid recognizing invalid computations and adding oracles that detect the invalid computations and flag them as errors. Both are cumbersome and add extra work to the software engineer writing the CALL specifications, hindering readability and simplicity of the CALL code.

A better way to declare the computation types is by defining additional elementary data types first and incorportating the constraints into those elementary data types. We start with the IMSI. The IMSI is a sequence of characters with at most 15 characters. CALL allows for the declaration of new data types derived from existing data types through a mechanism known as a *typedef*.

```
1  namespace telecom {
2
3  /*
4   * IMSIs are almost always 15 characters long,
5   * but some old IMSIs still in use are less.
6   */
7  const int32 k_max_imsi_length = 15;
8
9  /*
10  * Define the data type for an IMSI (International
11  * Mobile Subscriber Identifier), the unique ID for
12  * every mobile subscriber.
13  */
14 typedef imsi_t restricting string {
15      invariant max_length { length(self) <= k_max_imsi_length }
16 }
17
18 }
```

This code snippet defines a constant, `k_max_imsi_length` and defines a new data type, `imsi_t` that is a restriction on a `string`.

Typedefs are an important concept in CALL. A typedef is a construct that defines a data type that can be seen as a "copy" of the original data type with further restrictions (invariants). If `A` is a data type and `B` a typedef based on `A`, then the values of `B` are a subset of the values of `A`.

CALL allows implicit conversion from `imsi_t` to `string`, but requires explicit conversion the other way around.

The following snippet demonstrates implicit and explicit conversion between the `imsi_t` and `string`:

```
1 // From a string to an imsi_t requires explicit conversion:
2 imsi_t foo = "123";                    // Compilation fails.
3 imsi_t foo2 = convert<imsi_t>("123");  // OK
4 string foo3 = foo2;                    // OK
5 string foo4 = "012345678901234567890"; // OK
6 imsi_t foo5 = convert<imsi_t>(foo4);   // Compilation OK, fails a runtime.
```

Explicit conversion with typedefs is required as CALL checks the invariants at run time and the conversion may fail. On the other hand, conversion from the typedef to the type it is restricting will never break invariants.

Once we have defined the `imsi_t` we can now define some data types that allow reuse between the computation types.

We define a structure that contains the IMSI and a subtype of that structure that adds usage information. We call them `subscriber_data_t` and `subscriber_data_use_t`. These more general types can be used in functions that only use subscriber data.

```
namespace telecom {

/*
 * Structure with subscriber information. In the real
 * world this would contain more useful information,
 * but here we only have the IMSI.
 */
struct subscriber_data_t {
    imsi_t imsi;
}

/*
 * Structure with subscriber data usage. It
 * contains subscriber data in addition to usage.
 */
class subscriber_data_use_t : subscriber_data_t {
    invariant usage_non_negative { usage >= 0 }
    int32 usage;
}

}
```

CALL contains two types of data structures: `structs` and `classes`. Structures in CALL contain no logic: they are simple aggregators for properties. Classes can contain data and operations, although no operations are defined in this example. Classes can be subtypes of structures, but not the other way around. Structures are used, by convention, to represent data records, while classes are used when operations on data are required. This convention is close to the C++'s convention on structures and classes. However, there are important differences between CALL and C++: in C++ structures can contain methods, while in CALL they cannot. Also, in C++ there are differences in access constraints (structure's fields and methods are public by default, where on classes they are private), while CALL has no support for access constraints.

There are some choices that are worth noting: we chose to use `usage` as an `int32` and put the invariant in the class. Alternatively, we could have used a `typedef` to declare a non-negative integer data type. Also, `subscriber_data_t` was declared as a `struct` because no features from `class` are needed, although it could have been declared as a `class` instead.

Finally, we define *computation types* – the actual data that is probed from the system. computation types can be subtypes of structures, classes or other computation types. Because all properties were already defined in `subscriber_data_t` and `subscriber_data_use_t`, the computation types in this example are trivial: they extend the structures / classes, but they do not add additional properties, invariants or methods.

```
namespace telecom {

/*
 * CCR-I, CCR-U and CCR-T data types.
 */
computation type ccr_i_t : subscriber_data_t {}
computation type ccr_u_t : subscriber_data_use_t {}
computation type ccr_t_t : subscriber_data_use_t {}

/*
 * CCA response.
 */
```

```
13 computation type cca_t : subscriber_data_use_t {}
14
15 /*
16  * CDR.
17  */
18 computation type cdr_t : subscriber_data_use_t {}
19
20 }
```

4.2.2 Recognizers

In CALL, recognizers map lower-level computations into higher-level computations. They are used (as in Figure 3.1) to process streams of observations by recognizing patterns that can then be abstracted into a higher-level vocabulary. In their simplest form, recognizers contain an invariant (the matching condition) and an `emit` clause that defines the high-level computation to generate when the invariant is verified.

The general structure of a recognizer is:

```
1 recognizer type recognizer_name(i1 : in_comp_type_1, i2 : in_comp_type_2) {
2     invariant { some_condition_on_i1_and_i2 }
3     emit new out_comp_type(...);
4 }
```

The goals on the running example are to (1) verify that each CCR request is matched by a CCA, (2) verify that the CCRs that come after a CCA do not exceed the CCA's usage limit, and (3) emit a CDR. A CDR is a computation that defines a call: it is bounded in time by a CCR-I and a CCR-T that define the start and end of the call. The CDR contains usage data that is obtained by adding all CCR-Us and the CCR-T that occurred during the call.

Verifying CCR-CCA Match

We start by addressing issue (1) by writing a naïve recognizer that matches a CCR-I with a CCR-T and emits an intermediary computation type named `ccr_i_cca_t`.

```
1 computation type ccr_i_cca_t : subscriber_data_use_t {}
2
3 recognizer type ccr_i_cca_match_r(ccri : ccr_i_t, cca : cca_t) {
4     invariant { ccri.imsi == cca.imsi }
5     emit new ccr_i_cca_t(imsi: ccri.imsi, usage: 0);
6 }
```

This recognizer will map each CCR-I with a CCA as long as they have the same IMSI and outputs a CCR-I-CCA (the temporary structure) with the IMSI and 0 bytes usage.

The recognizer, however, will not work as intended (hence named naïve). Consider the following sequence of computations:

```
1 1.0 CCR-I (IMSI = 1)               [probed]
2 2.0 CCA   (IMSI = 1, usage = 100)  [probed]
3 3.0 CCR-I (IMSI = 1)               [probed]
4 4.0 CCA   (IMSI = 1, usage = 50)   [probed]
```

With this recognizer, the full sequence of computations, including those recognized would be:

```
1.0 CCR-I     (IMSI = 1)                [probed]
2.0 CCA       (IMSI = 1, usage = 100)   [probed]
2.1 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 1.0 and 2.0]
3.0 CCR-I     (IMSI = 1)                [probed]
3.1 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 3.0 and 2.0]
4.0 CCA       (IMSI = 1, usage = 50)    [probed]
4.1 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 1.0 and 4.0]
4.2 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 3.0 and 4.0]
```

The problem here is that this recognizer does not take into consideration that we want to match CCR-Is with CCAs such that (1) the CCA must come after the CCR-I and (2) no CCR-Is or CCAs must exist between the two recognized ones. The first condition would prevent the computation at 3.1 to be recognized, and the second condition would prevent the computation at 4.1 to be recognized. This is where quantifiers are required in recognizers.

The new recognizer would be:

```
computation type ccr_i_cca_t : subscriber_data_use_t {}

recognizer type ccr_i_cca_match_r(ccri : ccr_i_t, cca : cca_t) {
    invariant {
        ccri.imsi == cca.imsi
        and end(cca) > end(ccri)
        and not (exists
            t : &ccri @
                t.imsi = cca.imsi
                and end(t) > end(ccri)
                and end(t) < end(cca))
        and not (exists
            t : &cca @
                ccri.imsi == t.imsi
                and end(t) > end(ccri)
                and end(t) < end(cca))
        )
    }
    emit new ccr_i_cca_t(imsi: ccri.imsi, usage: 0);
}
```

This new definition of the recognizer introduces a more complex invariant that contains two important features:

- Use of the `end` function. Every computation has a time span associated with it. When a computation is recognized from a set of computations, it inherits the earliest start time and the latest end time in the set. The `end` function extracts the end time of a computation.

  The `end` function is used more often than the `start` function because computations are only available to the recognizers after the `end` time has elapsed. Before that they are still being computed. The `start` time can be used, for example, in oracles that assess how long a computation took to perform.

- Use of the `exists` operator. The `exists` operator is the equivalent to the ∃ operator in logic and evaluates over all computations of the given type. The `&` operator means "all computations of the type of `ccri`". This description is a simplification of the actual semantics, but, until a more accurate definition is provided in Chapter 5, we'll stick with this one.

  The `&` operator places no time bounds on the computations, so `t : &cca` means all computations of type `cca_t` that have ever been detected or that will ever be detected". It may

look awkward to refer to a set that contains future computations, which we don't know much about. But in CALL we're not defining the matching algorithms, we're defining relations between the computations.

For the moment, it may be surprising that CALL is actually capable of evaluating such an invariant, but more details will be provided in Chapter 5.

This recognizer would now produce a different sequence of computations:

```
1.0 CCR-I     (IMSI = 1)                [probed]
2.0 CCA       (IMSI = 1, usage = 100)   [probed]
2.1 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 1.0 and 2.0]
3.0 CCR-I     (IMSI = 1)                [probed]
4.0 CCA       (IMSI = 1, usage = 50)    [probed]
4.2 CCR-I-CCA (IMSI = 1, usage = 0)     [recognized by ccr_i_cca_t_match_r from 3.0 and 4.0]
```

As a side comment, we would like to emphasize that these matching constructs are simpler in CALL when compared to what would have to be written in other general purpose programming languages: there is no need to specify caches, buffers or any matching algorithms: a declarative logic statement is enough to express everything needed. Also, when comparing with complex event processing systems or relational databases, there is no need to define time windows, retention policies or user-defined functions (see Section 7.4).

This recognizer matches the correct computations and produces the expected output. Before moving to the other recognizers, it is easy to see that a significant part of the recognizer will be general to any call/return interaction: matching a call with the return that comes right after it. To extract the general call/return recognizer and see how reuse works in CALL, we'll follow a few steps.

Looking at the definition of `ccr_i_cca_match_r` we see that the match condition between a CCR-I and CCA (`ccri.imsi == cca.imsi`) is used multiple times. In this case, the predicate is simple, but, were the predicate a complex condition, it would be error-prone to repeat it manually; it would also add complexity to the expression. We will extract it to a separate class as a method.

This would yield the following definition:

```
1
2  computation type ccr_i_cca_t : subscriber_data_use_t {}
3
4  class ccr_i_cca_match_cond {
5      expr bool match(ccri : ccr_i_t, cca : cca_t) {
6          return ccri.imsi == cca.imsi;
7      }
8  }
9
10 recognizer type ccr_i_cca_match_r(ccri : ccr_i_t, cca : cca_t) {
11     cond : ccr_i_cca_match_cond;
12
13     init() {
14         cond = new ccr_i_cca_match_cond();
15     }
16
17     invariant {
18         cond.match(ccri, cca)
19         and end(cca) > end(ccri)
20         and not (exists
21             t : &ccri @
22                 cond.match(t, cca)
23                 and end(t) > ccri
```

```
                and end(t) < cca)
        and not (exists
            t : &cca @
                cond.match(ccri, t)
                and end(t) > ccri
                and end(t) < cca)
        )
    }
    emit new ccr_i_cca_t(imsi: ccri.imsi, usage: 0);
}
```

The `ccr_i_cca_match_cond` class defines the matching method. This method is marked with the `expr` keyword to signal it is an *expression method*. Expression methods are methods that can be inlined in recognizers and are limited in what they can contain: they can only declare local variables (but not reassign them) and return an expression. They can call other methods, but those methods must be expression methods themselves.

This recognizer's invariant is now more general as it doesn't contain the matching condition of two computations: it simply contains the ordering requirements. Continuing in this way, we could define more general computation types `call_t` and `return_t` and rewrite the class and recognizer to use the general data types. For reuse purposes, we place them in the `call_return` namespace:

```

namespace call_return {

computation type call_t {}
computation type return_t {}

class call_return_match {
    abstract bool match(c : call_t, r : return_t);
}

abstract recognizer type call_return_match_r(c : call_t, r : return_t) {
    cond : call_return_match;

    init(c : call_return_match) {
        cond = c;
    }

    invariant {
        cond.match(c, r)
        and end(r) > end(c)
        and not (exists
            cc : &c @
                cond.match(cc, r)
                and end(cc) > c
                and end(cc) < r)
        and not (exists
            rr : &cca @
                cond.match(c, rr)
                and end(rr) > c
                and end(rr) < r)
        )
    }
}

}
```

This recognizer is now marked as `abstract` as it does not contain an `emit` clause. This means we cannot directly use a `call_return_match_r`, but only use it as part of another

recognizer.

The `cci_i_cca_match_r` recognizer can now be defined in a simpler way, but the computations, CCR and CCA need to be also marked as call/return.

```
namespace telecom {

/*
 * CCR-I, CCR-U and CCR-T data types.
 *
 * Note that these are now marked as being calls to use the call/return
 * style defined above.
 */
computation type ccr_i_t : subscriber_data_t, call_return::call_t {}
computation type ccr_u_t : subscriber_data_use_t, call_return::call_t {}
computation type ccr_t_t : subscriber_data_use_t, call_return::call_t {}

/*
 * CCA response.
 *
 * Note that this is now marked as being a return to use the call/return
 * style defined above.
 */
computation type cca_t : subscriber_data_use_t, call_return::return_t {}

/*
 * The match condition now extends call_return_match and receives inputs
 * of type call_t and return_t.
 */
class ccr_i_cca_match_cond : call_return::call_return_match {
    expr bool match(c : call_return::call_t, r : call_return::return_t) {

        // The implementation of the method now needs to cast the types
        // to CCR and CCA as the language has no mechanism to define
        // templates / generics.
        ccri_t ccri = cast<ccri_t>(c);
        cca_t cca = cast<cca_t>(r);
        return ccri.imsi == cca.imsi;
    }
}

/*
 * The recognizer is now much simpler and makes use of the abstract
 * call/return recognizer.
 */
recognizer type ccr_i_cca_match_r(ccri : ccr_i_t, cca : cca_t) {
    // Declare we're using a call/return matcher as part of this one.
    recognizer cr : call_return_match_r;

    init() {
        // Initialize the call/return recognizer with the matching condition.
        init cr(new ccr_i_cca_match_cond());
    }

    // Just rely on call_return_match_r to do the recognition.
    invariant {
        cr(ccri, cca)
    }

    emit new ccr_i_cca_t(imsi: ccri.imsi, usage: 0);
}

}
```

Running the CALL compiler on the listing above will generate a warning that "the recognizer's eventual value stabilization time is infinite for parameter `ccri`". The reason for this

warning is an analysis performed by the CALL compiler on the recognizer's invariant. While the details of this analysis are described in Chapter 5, we can explain intuitively this result. If a CCR-I is detected and no other CCR-I or CCA is ever detected with the same IMSI, then the recognizer will have to hold this CCR-I computation indefinitely while waiting for a CCA.

This is a more general form of a common issue in call/return systems: if a call is issued and no response is obtained, the caller has to wait forever. While this may be the case in some systems, often to avoid indefinitely blocking of the caller there is some kind of timeout. In our case, we will assume that a CCA needs to be answered in less than 10 seconds. This requires slightly changing the match function:

```
namespace telecom {

const period k_cca_timeout = 10s;

/*
 * The match condition matches CCR-Is with CCAs (using generic call/return
 * types) that have the same IMSI and are within the given timeout.
 */
class ccr_i_cca_match_cond : call_return::call_return_match {
    expr bool match(c : call_return::call_t, r : call_return::return_t) {

        //  The implementation of the method now needs to cast the types
        // to CCR and CCA as the language has no mechanism to define
        // templates / generics.
        ccri_t ccri = cast<ccri_t>(c);
        cca_t cca = cast<cca_t>(r);
        return ccri.imsi == cca.imsi && end(r) < end(c) + k_cca_timeout;
    }
}

}
```

With this matching function, CALL will now know that any CCR-I without a response within 10s can no longer be matched to a CCA and can be dropped.

But this means we won't detect unmatched CCR-Is. And we want to detect these to signal incorrect behavior as all CCR-Is should have a response. This can be done using a specific recognizer that will flag CCR-Is for which no CCA match can be done:

```
namespace telecom {

computation type lost_ccr_i_t {
    ccri : ccr_i_t;
}

recognizer ccr_i_without_cca_r(ccri : ccr_i_t, &cca : cca_t) {
    recognizer ccri_a_match : ccr_i_cca_match_r;

    invariant {
        not (exists x : &cca @ ccri_a_match(ccri, x))
    }

    emit new lost_ccr_i_t(ccri: ccri);
}

}
```

This recognizer also relies on `ccr_i_cca_match_r`, but it has some differences: first we see in the signature it uses `cca_t &cca` instead of `cca_t cca`. The `&` symbol means "all CCAs" and this recognizer doesn't match CCR-Is with any CCAs. It is identifying CCR-Is, but

using the set of all CCAs to figure out which CCR-Is to identify. The ones to identify are exactly those that are not matched to any CCA.

Similarly, once we define recognizers that match CCR-U and CCR-T with CCA (in a similar way), we can define a recognizer that matches CCAs that are not matched to any CCR-I, CCR-U or CCR-T:

```
1  namespace telecom {
2
3  computation type lost_cca_t {
4      cca : cca_t
5  }
6
7  recognizer cca_without_ccr_r(cca : cca_t, &i : ccr_i_t, &u : ccr_u_t, &t : ccr_t_t) {
8      invariant {
9          not (exists ii : &i @ ccr_i_cca_match_r(ii, cca))
10         and not (exists uu : &u @ ccr_u_cca_match_r(uu, cca))
11         and not (exists tt : &t @ ccr_t_cca_match_r(tt, cca))
12     }
13
14     emit new lost_cca_t(cca: cca);
15 }
16
17 }
```

With these recognizers the system will:

- Match every CCR-I, CCR-U and CCR-T to their respective CCA; `ccr_i_cca_t`, `ccr_u_cca_t` and `ccr_t_cca_t` computations are generated;
- Identify CCR-Is, CCR-Us and CCR-Ts that did not have a CCA as a response; `lost_ccr_i_t`, `lost_ccr_u_t` and `lost_ccr_t_t` computations are generated;
- Identify CCAs which are not responses to any CCR-I, CCR-U or CCR-T; `lost_cca_t` computations are generated.

Checking CCA Usage Limit

In the context of the example, the intent is that CCRs and CCAs are used in pairs and form a flow. Each CCR requests credit information and each CCA describes how much data can be used. The CCR sent after the CCA will inform how much was used and receives a response with the allowed usage. A typical interaction for a user with 1.5Mb left of data might go like:

```
Message | Usage | Description
CCR-I   |       | (ask initial credit)
CCA     | 1Mb   | (let the user use up to 1Mb)
CCR-U   | 500Kb | (user has used 500Kb)
CCA     | 1Mb   | (let the user use up to 1Mb)
CCR-U   | 800Kb | (user has used 800Kb)
CCA     | 200Kb | (let the user use up to 200Kb)
CCR-T   | 150Kb | (user has used 150Kb and closed the connection)
CCA     | 50Kb  | (OK, 50Kb can still be used if the user wants)
```

Technically, the last CCA's usage is not used (keep in mind this is a simplification of the real credit protocol).

In a correctly-functioning system, the CCR that follows a CCA must not report more usage than it was allowed. We do this by matching every CCR-U and CCR-T with the previous CCA.

```
1  namespace telecom {
2
3  computation type cca_ccr_u_t {
4      invariant allowed_non_negative { allowed >= 0 }
5      invariant used_non_negative { used >= 0 }
6
7      imsi : imsi_t;
8      allowed : int32;
9      used : int32;
10 }
11
12 recognizer type cca_ccr_u_match_r(cca : cca_t, ccru : ccr_u_t) {
13     invariant {
14         ccru.imsi == cca.imsi
15         and end(ccru) > end(cca)
16         and not exists (
17             a : &cca @
18                 a.imsi == ccru.imsi
19                 and end(a) < end(ccru)
20                 and end(a) > end(cca))
21     }
22     emit new cca_ccr_u_t(imsi: ccru.imsi, allowed: cca.usage, used: ccru.usage);
23 }
24
25 }
```

Similar code could be written to match a CCA with the following CCR-T.

This recognizer may look similar to the one used to match a CCR-U with the following CCA, and this is not a coincidence. From a usage control perspective, the call/return could be seen reversed with the CCR-Us and CCR-Ts being a response to a CCA.

Differently from what we would do in a real specification where maintainability is important, we decided not to reuse the call/return recognizer to provide further insight into CALL. Instead, we have used a broader matching rule: we match a CCA with the following CCR-U, but we do not cater for the case in which two CCR-Us come in sequence. For example:

```
1.0 CCR-I      (IMSI = 1)                              [probed]
2.0 CCA        (IMSI = 1, usage = 100)                 [probed]
3.0 CCR-U      (IMSI = 1, usage = 50)                  [probed]
3.1 CCA-CCR-U  (IMSI = 1, allowed = 100, used = 50)    [recognized]
4.0 CCR-U      (IMSI = 1, usage = 70)                  [probed]
4.1 CCA-CCR-U  (IMSI = 1, allowed = 100, used = 70)    [recognized]
```

In this sequence, the CCR-U at 4.0 is matched with the CCA at 2.0 even though this same CCA was used to match the CCR-U at 3.0. We don't do this check as the protocol violation would be caught by the earlier recognizers. The need to keep track of what all recognizers are doing is one of the reasons we would not take this approach in a real system.

With these recognizers we would recognize computations `cca_ccr_u_t` and `cca_ccr_t_t`, but there is nothing (so far) stating that these computations represent incorrect behavior if the usage exceeds the allowed amount: that logic is part of the oracles, not the recognizers.

Grouping all CCRs into a CDR

With the final recognizers we want to build match a CCR-I with a CCR-T and with all CCR-Us in between. The desired output is a single computation, a CDR, that contains the total usage during a session.

This set of recognizers illustrate a limitation in CALL that will be further discussed in Section 9.1: the inability to match arbitrary groups of computations in a single computation. So, instead of directly generating a `cdr_t` from a `ccr_i_t`, a `ccr_u_t` and all `ccr_u_t`s between them, we need to do this in pairs of computations: if we have a sequence i, u_0, u_1, t, we recognize x_0 from i, x_1 from (x_0, u_0), x_2 from (x_1, u_1) and the CDR from (x_2, t).

```
1  namespace telecom {
2
3  computation type cdrtmp_t : subscriber_data_use_t {}
4
5  /*
6   * Recognizer that emits a cdrtmp_t from a ccr_i_t.
7   */
8  recognizer type ccr_i_to_tmp_r(ccri : ccr_i_t) {
9      invariant { true }
10     emit new cdrtmp_t(imsi: ccri.imsi, usage: 0);
11 }
12
13 /*
14  * Adds a CCR-U to a cdrtmp_t. The CCR-U is added if it
15  * has the same IMSI as the cdrtmp_t, and is located
16  * between "last" and the end of the cdrtmp_t.
17  */
18 recognizer type cdrtmp_ccr_u_r(tmp : cdrtmp_t, ccru : ccr_u_t) {
19     invariant {
20         ccru.imsi == tmp.imsi
21         and end(ccru) > end(tmp)
22         and (not exists u : &ccru @
23             u.imsi = tmp.imsi
24             and end(u) < end(ccru)
25             and end(u) > end(tmp))
26     }
27
28     emit new cdrtmp_t(imsi: tmp.imsi, usage: tmp.usage + ccru.usage);
29 }
30
31 /*
32  * Recognizer that emits a CDR when no more CCR-Us are
33  * available to add to the cdrtmp_t.
34  */
35 recognizer type cdrtmp_to_cdr_r(tmp : cdrtmp_t, &ccru : ccr_u_t, ccrt : ccr_t_t) {
36     invariant
37         tmp.imsi = ccrt.imsi
38         and end(tmp) < end(ccrt)
39         not (exists uu : &ccru @
40             uu.imsi = tmp.imsi
41             and end(uu) > end(tmp)
42             and end(uu) < end(ccrt));
43     emit new cdr_t(imsi: tmp.imsi, usage: tmp.usage);
44 }
```

4.2.3 Oracles

Recognizers, such as those presented in the previous section, are used to infer which computations have occurred based on observation of other computations. For example, as we have seen in the previous section, a recognizer can look at a CCR-I and a CCA and infer that a credit request/response has occurred.

As was described in Chapter 3, recognizers identify computations, but do not state whether they represent something good or something bad. The `cca_without_ccr_r` recognizer identifies CCAs without any matching CCR. These computations were named, in the example, `lost_cca_t`. But they do not say whether a lost CCA is a good or bad thing. That is the role of oracles.

Oracles are classifiers that look at computations and decide whether they represent a failure or not. The minimal form of an oracle is an invariant over a computation type: the invariant is validated if and only if the computation represents correct behavior.

In the telecommunications example that we have been using, we want to perform the following classifications:

- The total time from CCR to CCA is below a threshold of 10 seconds.
- All CCRs have a CCA as a response.
- There are no CCAs that are not a response to a matching CCR.
- The CCA usage limit is never exceeded.

The code snippet below defines an oracle that evaluates the total time from CCR-I to CCA.

```
1  namespace telecom {
2
3  const period k_max_cca_response = 10s;
4
5  oracle type ccr_i_cca_response_time_o(c : ccr_i_cca_t) {
6      invariant {
7          end(c) - start(c) <= k_max_cca_response
8      }
9  }
10
11 }
```

Oracles have an invariant that contains a predicate evaluating the computation. Each oracle type evaluates computations of exactly one type or its subtypes.

Oracles, like recognizers, can depend on other oracles to perform their evaluation. While the feature is not strictly needed given the simplicity of this oracle, we could factor out part of this oracle into an abstract oracle that places a limit on the time span of a computation:

```
1  namespace telecom {
2
3  /*
4   * Declare an abstract oracle that depends on the generic
5   * type computation_t. This type is predefined and all
6   * computations inherit from it even if not explicitly
7   * declared.
8   */
9  abstract oracle type limit_o(c : computation_t) {
10     limit : period;
11
12     init(l : period) {
13         limit = l;
```

```
    }

    invariant {
        end(c) - start(c) < limit;
    }
}

const period k_max_cca_response = 10s;

oracle type ccr_i_cca_response_time_o(c : ccr_i_cca_t) {
    // Declare we're using an oracle.
    oracle l : limit_o;

    // Initialize the oracle. Non-abstract oracles cannot
    // have initialization parameters.
    init() {
        init l(k_max_cca_response);
    }

    // Our invariant delegates evaluation to the abstract
    // oracle.
    invariant {
        l(c);
    }
}

}
```

This structure is similar to what we did with factoring out the `call_return_match_r` recognizer. Similar oracles could be defined that would evaluate the time taken to send a CCA as a response to CCR-Us or CCR-Ts.

Another set of required oracles in our example are the oracles that state that all `lost_` computations represent failures. These oracles are common in CALL specifications as they state that some computations should never happen. They have trivial invariants:

```
namespace telecom {

oracle type lost_ccr_i_o(c : lost_ccr_i_t) {
    invariant { false }
}

}
```

Similar oracles would be built for the other computation types that represent unmatched CCRs or CCAs.

Lastly, we need to define the oracles that state that usage stays within limits. This is trivially done by evaluating the data present in the `cca_ccr_u_t` and `cca_ccr_t_t` computation types (example below for `cca_ccr_u_t`):

```
namespace telecom {

oracle type limit_ccr_u(c : cca_ccr_u_t) {
  invariant { c.used <= c.allowed }
}

}
```

A similar oracle would be defined for `cca_ccr_t_t`.

In general, oracles tend to be much simpler than recognizers: quantifiers are not available, they operate on a single computation at a time and, consequently, they do not have to perform

pattern matching and, because they evaluate high-level computations, it is usually simple to state whether a computation represents correct behavior.

4.2.4 Streams

Computations in CALL are managed in sets called *streams*. A stream is a set of computations of the same type. Because computations are timed, at any point in time, we can split the set into an observed part and an unobserved part. The observed part contains all computations that have finished before the current time and the unobserved part contains all computations that are not observable. As time progresses, we "see" computations appearing in the observed part of the set.

The observed part of the set is obviously needed when CALL is executing the recognizers. The unobservable part may not seem as critical, but it is often a fundamental part of pattern matching. For example, consider a system with two computation types, a *ping* and a *pong*. A recognizer that matches a lost ping as "a ping without a pong being received in 5 seconds could look like":

```
1  computation type ping_t {
2      id : int32;
3  }
4
5  computation type pong_t {
6      id : int32;
7  }
8
9  computation type missing_pong_t {
10     id : int32;
11 }
12
13 recognizer type ping_pong_r(ping : ping_t, &pong : pong_t) {
14     invariant {
15         ping.id = pong.id
16         and not (exists p : &pong @
17             ping.id = p.id
18             and end(p) < end(ping) +5s)
19     }
20
21     emit new missing_pong_t(id: ping.id);
22 }
```

This recognizer is attached to three streams, one of type `ping_t`, another of type `pong_t` and another of type `missing_pong_t`. Let's consider the scenario where a ping is missing a pong.

The streams in this case could be:

```
ping_t = { Ping 1 at 1s, Ping 2 at 3s }
pong_t = { Pong 2 at 7s }
missing_pong_t = { Missing 1 at 6s }
```

The recognizer `ping_pong_r` establishes a relation between these three streams. The reason why `Missing 1` is computed at time 6s is explained by the unobserved parts. At 4s, the observed part of the `pong_t` stream is empty. While there will be no match for the ping at 1s, there is no way for the recognizer to know that at this time since a pong could exist in the unobserved part of `pong_t`. However, at 6s – and only at 6s –, there cannot be any matches for

ping 1 in the unobserved part of `pong_t` because those would have an end time greater than 6s and they wouldn't match the invariant.

Splitting streams into observed and unobserved, and reasoning about what can and cannot be in the unobserved part of streams is not present in any of the traditional Complex Event Processing systems. Unlike these systems, CALL does not require a definition of time windows and can simultaneously handle large volumes of computations that can be quickly discarded while keeping a small volume of computations that are needed for a long time. See Section 7.4 for a more detailed comparison with Complex Event Processing systems.

Streams, as described above, form the wiring of a CALL program: probes, recognizers and oracles are all connected through streams. Connecting a probe, recognizer or oracle to a stream is termed "binding". Each input and output of probes, recognizers or oracles has to be bound to a stream.

In the telecommunications example being used, we declare several streams as shown below.

```
1  namespace telecom {
2
3  // Streams that contain computations detected in
4  // the target system.
5  stream ccr_i_s : ccr_i_t;
6  stream ccr_u_s : ccr_u_t;
7  stream ccr_t_s : ccr_t_t;
8  stream cca_t_s : cca_t;
9
10 // Streams that contain matched CCRs with CCAs.
11 stream ccr_i_cca_s : ccr_i_cca_t;
12 stream ccr_u_cca_s : ccr_u_cca_t;
13 stream ccr_t_cca_s : ccr_t_cca_t;
14
15 // Streams that contain CCAs matched to their
16 // subsequent CCR-Us and CCR-Ts.
17 stream cca_ccr_u_s :  cca_ccr_u_t;
18 stream cca_ccr_t_s :  cca_ccr_t_t;
19
20 // Streams with unmatched CCRs and CCAs.
21 stream lost_ccr_i_s :  lost_ccr_i_t;
22 stream lost_ccr_u_s :  lost_ccr_u_t;
23 stream lost_ccr_t_s :  lost_ccr_t_t;
24 stream lost_cca_s :  lost_cca_t;
25
26 // Stream with temporary computations for CDRs.
27 stream cdrtmp_s : cdrtmp_t;
28
29 // Stream with CDRs.
30 stream cdr_s : cdr_t;
31 }
```

In this example we can see a pattern that is common, but not required by CALL: there is a 1 to 1 relationship between computation types and streams. While each stream must contain computations of a single type (or its subtypes), there is no requirement that each computation type be used in just one stream.

The declarations in the CALL specification above just state what the streams are, they don't state how they are bound to recognizers. We won't be doing that here.

4.2.5 Runtime Model

So far we've described how to declare computation types, recognizer types, oracle types and streams. These are the concepts of CALL that directly represent the approach architecture described in Section 3. In this section we discuss the runtime model of CALL: where (physically) the computations are performed.

The runtime model is a fundamental part of the design of CALL as it supports scalability by splitting execution of the various tasks over a set of processes. Scalability is one of the claims made in the thesis statement in Chapter 1 as the framework needs to handle thousands of computations per second (see case studies in Section 8.1).

In previous sections we've introduced recognizer and oracle *types*, but have not shown any CALL statements that state which *instances* exist. This is done in the runtime model. In the following listing, we instantiate the recognizers and oracles:

```
1  recognizer ccr_i_cca_match_ri : ccr_i_cca_match_r;
2  recognizer ccr_u_cca_match_ri : ccr_u_cca_match_r;
3  recognizer ccr_t_cca_match_ri : ccr_t_cca_match_r;
4  recognizer ccr_i_without_cca_ri : ccr_i_without_cca_r;
5  recognizer ccr_u_without_cca_ri : ccr_u_without_cca_r;
6  recognizer ccr_t_without_cca_ri : ccr_t_without_cca_r;
7  recognizer cca_without_ccr_ri : cca_without_ccr_r;
8  recognizer cca_ccr_u_match_ri : cca_ccr_u_match_r;
9  recognizer cca_ccr_t_match_ri : cca_ccr_t_match_r;
10 recognizer ccr_i_to_tmp_ri : ccr_i_to_tmp_r;
11 recognizer cdrtmp_ccr_u_ri : cdrtmp_ccr_u_r;
12 recognizer cdrtmp_to_cdr_ri : cdrtmp_to_cdr_r;
13
14 oracle ccr_i_cca_response_time_oi : ccr_i_cca_response_time_o;
15 oracle ccr_u_cca_response_time_oi : ccr_u_cca_response_time_o;
16 oracle ccr_t_cca_response_time_oi : ccr_t_cca_response_time_o;
17 oracle lost_ccr_i_oi : lost_ccr_i_o;
18 oracle lost_ccr_u_oi : lost_ccr_u_o;
19 oracle lost_ccr_t_oi : lost_ccr_t_o;
20 oracle lost_cca_oi : lost_cca_o;
21 oracle limit_ccr_u_oi : limit_ccr_u_o;
22 oracle limit_ccr_t_oi : limit_ccr_t_o;
```

In general, there will be one recognizer and one oracle per recognizer type and oracle type, respectively. However, just like streams, we can instantiate several recognizers or oracles with the same type. This is useful for large systems where recognizers and oracles can run in localized parts of the system. For example, a specification can declare recognizer type that recognize invalid HTTP computations, but then use one recognizer per server.

After instantiating recognizers and oracles we need to bind them to streams:

```
1  bind ccr_i_cca_match_ri in (ccr_i_s, cca_s) out ccr_i_cca_s;
2  bind ccr_u_cca_match_ri in (ccr_u_s, cca_s) out ccr_u_cca_s;
3  bind ccr_t_cca_match_ri in (ccr_t_s, cca_s) out ccr_t_cca_s;
4  bind ccr_i_without_cca_ri in (ccr_i_s, cca_s) out lost_ccr_i_s;
5  bind ccr_u_without_cca_ri in (ccr_u_s, cca_s) out lost_ccr_u_s;
6  bind ccr_t_without_cca_ri in (ccr_t_s, cca_s) out lost_ccr_t_s;
7  bind cca_without_ccr_ri in (cca_s, ccr_i_s, ccr_u_s, ccr_t_s) out lost_cca_s;
8  bind cca_ccr_u_match_ri in (cca_s, ccr_u_s) out cca_ccr_u_s;
9  bind cca_ccr_t_match_ri in (cca_s, ccr_t_s) out cca_ccr_t_s;
10 bind ccr_i_to_tmp_ri in (ccr_i_s) out cdrtmp_s;
11 bind cdrtmp_ccr_u_ri in (cdrtmp_s, ccr_u_s) out cdrtmp_s;
12 bind cdrtmp_to_cdr_ri in (cdrtmp_s, ccr_u_s, ccr_t_s) out cdr_s;
13
14 bind ccr_i_cca_response_time_oi in (ccr_i_cca_s);
```

```
15 bind ccr_u_cca_response_time_oi in (ccr_u_cca_s);
16 bind ccr_t_cca_response_time_oi in (ccr_t_cca_s);
17 bind lost_ccr_i_oi in (lost_ccr_i_s);
18 bind lost_ccr_u_oi in (lost_ccr_u_s);
19 bind lost_ccr_t_oi in (lost_ccr_t_s);
20 bind lost_cca_oi in (lost_cca_s);
21 bind limit_ccr_u_oi in (cca_ccr_u_s);
22 bind limit_ccr_t_oi in (cca_ccr_t_s);
```

These declarations bind all recognizers and oracles to the streams previously declared. This step is known as *wiring*. At this point, the CALL runtime knows what recognizers and streams to instantiate and how to connect them.

4.2.6 Deployment

If the CALL runtime were to execute in a single process, the wiring would be enough to run a CALL program. However, because CALL supports distributed execution we need to specify how deployment is done.

Deployment is written in CALL by describing which *nodes* exist. Each node will correspond to a process. When starting the CALL runtime, the name of the node is provided so that the process knows which node it is running.

Below is a node description for the telecommunications example we've been following.

```
1  node rec_n {
2    recognizer ".*";
3  }
4
5  node orc_n {
6    oracle ".*";
7  }
8
9  node fl_n {
10   fault_localizer;
11 }
12
13 node bus_n {
14   ip "10.0.0.5";
15   event_bus {
16     port 5456;
17     stream ".*";
18   }
19 }
```

This listing declares 4 nodes. The first will run all of the recognizers (`.*` is a regular expression that matches all recognizer names). The second will run all of the oracles and the third will run the fault localizer. There has to be one and only one fault localizer.

The fourth node runs the event bus through which all other nodes send and receive computations. The event bus contains all of the streams. Recognizers and oracles can be split into any number of nodes. If the number of computations to be handled is too large for a single event bus to route all of them to the recognizers, several event buses can be run to distribute the computations.

From a networking perspective, it is necessary that all recognizers and oracles can connect to the nodes running the event buses that contain the streams they read from or write to. The oracles will publish the result of evaluating computations to the same node they read the computations

from. The fault localizer will read the computations from the event buses that contain streams wired to oracles.

Chapter 5

CALL formal semantics

In this section we present a formal semantics for the CALL language. Two complimentary semantics are provided: a *denotational semantics* (Section 5.1) in which we give formal meaning to the language constructs disregarding how they are actually implemented and an *operational semantics* (Section 5.2) in which we describe the algorithms used to implement the language.

5.1 Denotational Semantics

In this section we will describe what is meant by the CALL's language constructs. We will describe these in terms of set theory. CALL is a language and the first section of the semantics, Section 5.1.2 provides the definition and structure of an CALL program. Section 5.1.3 describes the denotational semantics of CALL.

All CALL descriptions provided in this chapter are untyped, but CALL is a strictly typed language. However, the type system has no impact on the semantics on the language, it is provided to improve reusability and practicality. Section 5.2.7 describes CALL's type system and discusses why it can be safely discarded for the purposes of this chapter.

Some elements of CALL are not covered in this chapter as they bare no relevance for the semantics of the core part of language.

Section 5.1.1 is an introductory section providing an overview of the notation we're using and can be skipped and used as reference.

5.1.1 Notation and Terminology

We borrow most of the notation from logic and set theory. In the following subsections we present the notation and terminology used.

Types

Types are represented as $\mathbb{A}$. A type is a set. A value has a type if it belongs to the set. For example, value a has type $\mathbb{A}$ if and only if $a \in \mathbb{A}$. Each value belongs to only one type, meaning the intersection of any two types is empty.

We say a variable a has type $\mathbb{A}$ (and write $a : \mathbb{A}$) if the values the variable can have are elements of type $\mathbb{A}$.

Fundamental Types

There are four fundamental types:

- The empty type (or empty set), represented as $\varnothing$, a set with no elements.
- The boolean type ($\mathbb{B}$), containing the values true ($\top$) and false ($\bot$). Values drawn from this set are represented as q, r, ... see below for a discussion on propositional logic and first order logic for operations on boolean values.
- The integer type ($\mathbb{Z}$), containing all integer numbers. Throughout this chapter, the term integer and natural numbers are used interchangeably. Unless explicitly noted, no negative numbers are used. Integer numbers are represented as i, j, ... see below for a discussion on integer types.
- The real type ($\mathbb{R}$), containing all real numbers. See below for a discussion on real types.

Sets

Sets are represented as A. Sets can be represented by explicitly enumerating the element in the set. For example, a set with elements a_0 and a_1 can be written as $\{a_0, a_1\}$. A set with no elements is the empty set, represented as $\varnothing$.

We use the following set operations (most of which are standard):

- Presence test. $a \in B$ is true if and only if a belongs to set B.
- Absence test. $a \notin B$ is true if and only if a does not belong to set B.
- Power set. $\mathbb{P}A$ is the set of all subsets of A.
- Strict subset. $A \subset B$ is true if and only if A is a strict subset of B.
- Subset. $A \subseteq B$ is true if and only if A is a subset of B.
- Union. $A \cup B$ is the set with all the elements of both A and B.
- Intersection. $A \cap B$ is the set with all the elements that exist in A and in B.
- Difference. $A \backslash B$ is the set with all the elements that exist in A, but not in B.
- Full union. If A is a set of sets, then $\bigcup_A$ is the set with all elements in all the sets in A.

We define a range operator that builds a set of natural numbers. If i and j are two integer numbers such that $i \leqslant j$, then $i..j$ is the set of all natural numbers k such that $k \geqslant i$ and $k \leqslant j$.

We represent set comprehension as $\{a : A | b \mapsto c\}$ where A defines the values to run the comprehension on, a represents one of the values in A, b is the selection predicate and c the optional result transformation. In some cases we only use set comprehension as a filter and represent it as $\{a|b\}$, which implies an identity result transformation. Comprehension can also be used without filter and represent it as $\{a \mapsto b\}$, which implies all elements are transformed.

Integer and Real Types

Standard arithmetic operations are used for integer and real types:

- Addition: $i + j$.
- Subtraction: $i - j$.
- Multiplication: $i \times j$.
- Inversion: $-i$.
- Maximum selection: $\max A$, where A is a set of numbers.

The following comparisons are made between integer and real types:

- Equality: $i = j$.
- Inequality: $i \neq j$.
- Lesser than: $i < j$.
- Lesser than or equal to: $i \leqslant j$.
- Greater than: $i > j$.
- Greater than or equal to: $i \geqslant j$.

Tuples

A tuple is a type that is composed by an ordered list of other types. Each value of a tuple type contains one value from one of the composing types. A tuple is represented as a cartesian product. For example, the tuple type that has types $\mathbb{A}$ and $\mathbb{B}$ is written as $\mathbb{A} \times \mathbb{B}$.

A tuple value is an ordered list of other values, each value drawn from its corresponding type. We say (a, b) is a tuple value whose first value is a and second value b.

With this definition, $\mathbb{A} \times \varnothing = \varnothing$. While this a well known result from set theory, it may not be obvious that for the purposes of the semantics, we consider this to be also the case for types.

Vectors

A vector, sometimes called a sequence, is a type that is composed by an integer number (possibly zero) of elements of the same type. We represent a vector of $i : \mathbb{Z}$ elements of type $\mathbb{A}$ as $\mathbb{A}^i$.

We represent values of type vector as $\overrightarrow{a}$. We can also represent a vector by enumerating the elements in the vector. For example, if a_1 and a_2 are two values of type $\mathbb{A}$, then we can represent the vector value of type $\mathbb{A}^2$ that contains those two values as $[a_0, a_1]$.

Vectors are used extensively in the semantics and we define multiple functions to manipulate them:

- Number of elements in a vector. If $\overrightarrow{a}$ is a vector, then $\#\overrightarrow{a}$ is the number of elements in the vector.
- Obtaining an element of a vector. If $\overrightarrow{a}$ is a vector, then $a[i]$ obtains the value in position i of the vector. Positions are indexed from zero. If $i \geqslant \#\overrightarrow{a}$ the result is undefined. For example, if $\overrightarrow{a} = [a_0, a_1]$, then $\overrightarrow{a}[1] = a_1$.

- Appending. If $\overrightarrow{a} : \mathbb{A}^i$ and $b : \mathbb{A}$, then $\overrightarrow{a} + (b)$ is a value of type $\mathbb{A}^{i+1}$, for which all the positions before i have the same value as in $\overrightarrow{a}$ and $(\overrightarrow{a} + (b))[i] = b$.
- Prepending. If $\overrightarrow{a} : \mathbb{A}^i$ and $b : \mathbb{A}$, then $(b) + \overrightarrow{a}$ is a value of type $\mathbb{A}^{i+1}$, for which all positions j greater than zero have the same value as the $j - 1$ ones in $\overrightarrow{a}$ and $(\overrightarrow{a} + (b))[0] = b$.
- Inserting. If $\overrightarrow{a} : \mathbb{A}^i$, $b : \mathbb{A}$ and $j : \mathbb{Z}$, such that $j \leqslant i$, then $\overrightarrow{c} : A^{i+1}$, defined as $\overrightarrow{c} = \overrightarrow{a} +_j (b)$, is a vector such that, (1) for every $k : \mathbb{Z}$ such that $k < j$, $\overrightarrow{a}[k] = \overrightarrow{c}[k]$; (2) $\overrightarrow{c}[j] = b$; (3) for every $k : \mathbb{Z}$ such that $k > j$ and such that $k \leqslant i$, $\overrightarrow{c}[k] = \overrightarrow{a}[k-1]$.

  Both appending and prepending can be written in terms of inserting: $\overrightarrow{a} + (b) = \overrightarrow{a} +_{\#\overrightarrow{a}} (b)$ and $(\overrightarrow{a}) + b = \overrightarrow{a} +_0 (b)$. However, appending and prepending is done frequently and often the specific notation for appending and prepending is used.
- Removing. If $\overrightarrow{a} : \mathbb{A}^i$ and $j : \mathbb{Z}$, such that $j < i$, then the vector $\overrightarrow{c} : i - 1$ defined as $\overrightarrow{c} = \overrightarrow{a}_{-j}$ is a vector such that (1) for every $k : \mathbb{Z}$, such that $k < j$, $\overrightarrow{c}[k] = \overrightarrow{a}[k]$ and (2) for every $k : \mathbb{Z}$, such that $k \geqslant j$, $\overrightarrow{c}[k] = \overrightarrow{a}[k+1]$.
- Removing last. If $\overrightarrow{a} : \mathbb{A}^i$ and $i > 0$, then $\overrightarrow{a}_- = \overrightarrow{a}_{-(\#\overrightarrow{a})-1}$.
- Extracting range. If $\overrightarrow{a} : \mathbb{A}^i$, $j : \mathbb{Z}$, $k : \mathbb{Z}$ such that $j \leqslant k$ and $k < i$, then $\overrightarrow{b} : \mathbb{A}^{k-j+1}$ defined as $\overrightarrow{b} = \overrightarrow{a}_{j..k}$ is a vector such that for any $l : \mathbb{Z}$ such that $l < k - j + 1$, $\overrightarrow{c}[l] = \overrightarrow{a}[j + l]$.
- Appending vectors (concatenating). Let $\overrightarrow{a} : \mathbb{A}^i$ be a vector, $\overrightarrow{b} : \mathbb{A}^j$, be another vector, then the vector $\overrightarrow{c} : \mathbb{A}^{i+j}$, defined as $\overrightarrow{c} = \overrightarrow{a} + \overrightarrow{b}$, is a vector such that (1) for any $k : \mathbb{Z}$, $k < i$, $\overrightarrow{c}[k] = \overrightarrow{a}[k]$; (2) for any $k : \mathbb{Z}$, $k < j$, $\overrightarrow{c}[k + i] = \overrightarrow{b}[k]$.
- Vector internal cross. Let $\overrightarrow{a} : (\mathbb{P}\mathbb{A})^i$, then $\overrightarrow{b} : \mathbb{P}\mathbb{A}^i$, defined as $\overrightarrow{b} = \times(\overrightarrow{a})$, contains all combinations of elements in $\overrightarrow{a}$.
- For-each. Let $\overrightarrow{a} : \mathbb{A}^i$, f, be a function that maps $\mathbb{A}$ to $\mathbb{B}$ (see below for functions and predicates), then $b : \mathbb{B}^i$ defined as $\overrightarrow{b} = [c : \overrightarrow{a} \mapsto f(c)]$, is a vector such that for all $j : \mathbb{Z}$, $j < i$, $\overrightarrow{b}[j] = f(\overrightarrow{a}[j])$.

Functions and Predicates

Functions represent a relation between types. They map a type, its domain, to another type, its counter-domain. Functions are named as f, g, ... If $\mathbb{A}$ is a function's domain and $\mathbb{B}$ is the function's counter-domain, we represent the function's type as $\mathbb{A} \to \mathbb{B}$. Obtaining a specific mapping of a value in a function's domain is represented by standard function application notation: $f(i)$. The result of such application is a value from $\mathbb{B}$.

Functions may have multiple "arguments". This is the case when the domain is a tuple. For example, if a function f has domain $\mathbb{A} \times \mathbb{B}$, then we represent the mapping with $f(i, j)$.

Lambdas

A lambda is a function that also has a domain and counter-domain. Unlike functions, lambdas are not named and a formula for calculating the mapping value is provided. For example, a lambda that maps $\mathbb{R} \times \mathbb{Z}$ into $\mathbb{R}$ by computing the exponentiation can be written as $\lambda_{i:\mathbb{R},j:\mathbb{Z}}.i^j$.

A lambda defines a function, not an application of a function. The syntax for obtaining the mapping with a lambda is the same as the one with a function: for example, to obtain the mapping of values 5 and 6 in the lambda defined above, the syntax would be:

$$\left(\lambda_{i:\mathbb{R},j:\mathbb{Z}}.i^j\right)(5,6)$$

Logic

As described in Section 5.1.1, we represent the set of boolean values true ($\top$) and false ($\bot$) as $\mathbb{B}$. Logic operators follow the standard syntax:

- conjunction: $a \wedge b$
- disjunction: $a \vee b$
- equivalence: $a \Leftrightarrow b$
- implication: $a \Rightarrow b$
- negation: $\neg a$

We represent logic variables as: q, r, ... A formula made by applying the connectives above with variables is termed a *logic formula*. For example, the following is a logic formula:

$$q \vee (r \wedge s)$$

Logic formulas are used often with predicates instead of, or in addition to, variables. For example, for the following is a logic formula dependent on one variable and two predicates over i and j:

$$i \leqslant j \vee (q \wedge i = j)$$

We also use the standard quantifiers *for all* ($\forall$) and *exists* ($\exists$). At various points in this document it is necessary to distinguish between formulas with quantifiers and formulas without quantifiers and we will use the term *quantified formula* and *non-quantified formula*, respectively.

In logic formulas it is possible to quantify either sets or entire types due to the fact that we interpret types as sets (see Section 5.1.1). For example, the following first order logic formula evaluates to true if all sets in a set of sets of integer numbers (A) contain at least a prime number.

$$\forall_{B:\in A}\exists_{c\in b}\neg\exists_{d\in\mathbb{Z}}d \neq 1 \wedge d \neq c \wedge \exists_{e\in\mathbb{Z}}c = d \times e$$

Because a predicate is any function that evaluates to $\mathbb{B}$ (see Section 5.1.1), all logic formulas are predicates and the two terms are often used interchangeably when referring to logic formulas.

Time

For the purpose of the semantics, time is considered an infinite set. A few assumptions are made on this set:

- The set is infinite.
- The set is ordered.

- All operations below are defined for all values (except where noted).
- Just like for real numbers, we use $-\infty_t$ and $+\infty_t$ to denote infinitely backwards in the past or infinitely forward in the future.

Time is represented by $\mathbb{T}$. Time variables are represented as t, u, ...Neither boolean nor arithmetic operations are allowed on time. The following operations are defined:

- Before: $t <_t u$.
- Before or same: $t \leqslant_t u$.
- After: $t >_t u$.
- After or same: $t \geqslant_t u$.
- Addition: $t +_t u$. $-\infty_t +_t +\infty_t$ is not defined.
- Subtraction: $t -_t u$. $+\infty_t -_t +\infty_t$ and $-\infty_t -_t -\infty_t$ are not defined.
- Inversion: $-_t t$.

The time set, $\mathbb{T}$, because it contains two infinite values, can be thought of being equivalent to the Dedekind–MacNeille completion of $\mathbb{R}$, also known as the extended real number line, $\bar{\mathbb{R}}$.

Specific points in time are referred in seconds. For example, the time point of 5 seconds is represented as $5s$.

Intervals

An interval is a possibly empty range of time values with a lower and upper bounds, either of which can be infinity. The lower bound is never after the upper bound. Intervals may be closed or open meaning that the bound is included or not included in the interval. The set of all intervals (or the type of an interval) is $\mathbb{I}$. Intervals are notated as I, J, ...

Closed intervals are represented as $[t, u]$. Open intervals are represented as $]t, u[$. Intervals that are open on one end and closed on the other are represented as $[t, u[$ or $]t, u]$. $]t, t[$, $[t, t[$ and $]t, t]$ are all empty intervals.

Time values can be checked for belonging to the interval with $t \in I$. The negated version is $t \notin I$. The lower and upper bounds of an interval I can be obtained with $\mathrm{L}(I)$ and $\mathrm{U}(I)$, respectively. Intervals can be intersected as $I \cap_I J$. It is also possible to compute the union of two intervals: $I \cup_I J$: this results in an interval with the lowest bound of both intervals and the upper bound of both intervals.

Intervals can be shifted. Shifting is an operation that moves both bounds by a specific amount. For example, if interval I is $]t, u[$, shifting this interval by v, represented by $[I]^v$, results in interval $]t +_t v, u +_t v[$.

Predicates

A predicate is any function that maps its inputs to $\mathbb{B}$. Predicate functions can be built on propositional logic, in which case they are known as a propositional logic predicate, or they can be built on first order logic, in which they are known as first order logic predicates.

A predicate is a logic formula where:

- Individual terms are either boolean inputs or comparisons of non-boolean terms.

- Quantifiers may iterate over sets defined by inputs or non-boolean terms that are sets.
- Non-boolean terms can be inputs to the predicate or functions made of other non-boolean terms.

Logic Terminology

Throughout this document we will use these terms or the general term *logic* to refer to any type of logic, usually predicate logic or predicate logic with quantifiers. The term *predicate* – used interchangeably with *formula* – is used to refer to boolean predicates, as described earlier in this section.

Computations

The type of all computations is $\mathbb{C}$. We denote individual computations by letters: a, b and c. We denote sets of computations as A, B and C.

As will be described later, computations have typed attributes. We denote the attribute *foo* of a computation a as $\texttt{foo}(a)$.

5.1.2 CALL Programs

CALL is a language that deals with computations where low-level computations are probed from the system and are abstracted into higher level computations and then classified. A CALL program is a program written using the CALL language. Each program defines probes, recognizers and oracles as described in Chapter 3.

In a CALL program, there is a number of probes, recognizers and oracles. There is always only one fault localizer whose semantics are not relevant for the CALL language, although it is relevant for the thesis as a whole.

5.1.3 Untyped Denotational Semantics

A CALL program has an input and an output. The input is defined as a tuple with one set of computations per probe. These represent the computations that are observed by the system. The output is defined as a tuple with one set of classified computations (computations marked as correct or incorrect) per oracle. These represent the input to the fault recognizer.

More formally, if $\mathbb{C}$ is the set of all computations, then a CALL program P with $p_\#$ probes and $o_\#$ oracles can be represented as a function:

$$P : (\mathbb{P}\mathbb{C})^{p_\#} \rightarrow (\mathbb{P}\,(\mathbb{C} \times \mathbb{B}))^{o_\#}$$

The $(\mathbb{P}\mathbb{C})^{p_\#}$ input states that the program receives $p_\#$ sets of computations as input. Each set represents the computations detected by each one of the probes. The $(\mathbb{P}\,(\mathbb{C} \times \mathbb{B}))^{o_\#}$ output states that the program produces $o_\#$ sets of elements with type $\mathbb{C} \times \mathbb{B}$. Each set contains classified computations and corresponds to the output of an oracle. This definition assumes there is some ordering in the probes and in the oracles. This assumption is artificial as a reordering of probes

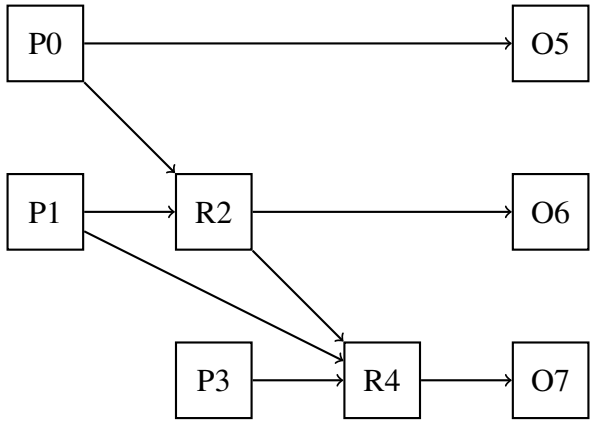


Figure 5.1: DAG representing an example CALL program. P nodes are probes, R nodes are recognizers and O nodes are oracles.

and/or oracles does not result in a different program. However, it is a simplifying assumption that does not significantly impact the semantics.

The program itself is represented as a directed acyclic graph (DAG) where each input node with no inputs is a probe, each node with no outputs is an oracle and all other nodes are recognizers. A program with $p_{\#}$ probes will therefore have $p_{\#}$ input nodes and a program with $o_{\#}$ oracles will have $o_{\#}$ output nodes. Output nodes can only have one incoming edge.

Figure 5.1 has the DAG for an example CALL program.

Recognizers' incoming edges determine which computation sets are made available to the recognizer. Each incoming edge from a probe makes the probed computations available to the recognizer and each incoming edge from a recognizer makes the latter recognizer's output (also a set of computations) available to the former. Each recognizer has a single output that is made available to all oracles and recognizers it is connected to.

Each recognizer produces a single output set (the "output") from all incoming sets. Incoming sets are split into two groups: the "inputs" and the "context". Although not represented in the graph in Figure 5.1, this split can be represented as labeling on the graph's edges that lead to recognizers. A recognizer can be defined as a function that maps all inputs in the presence of context to the output. Therefore a recognizer r with i input sets and j context sets has type:

$$r : (\mathbb{PC})^i \times (\mathbb{PC})^j \to \mathbb{PC} \tag{5.1}$$

Equation 5.1 is a general formula, but not all functions that have this type can be recognizers. Any recognizer r with i input sets and j context sets must be formed by two different functions: a selection predicate f and an emission function e that have the types defined in Equation 5.2 and Equation 5.3.

$$f : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B} \tag{5.2}$$

$$e : \mathbb{C}^i \to \mathbb{C} \tag{5.3}$$

There are further restrictions for the selection predicate. It must be a propositional logic predicate that may have quantifiers over the sets in $\overrightarrow{B}$. The sets can *only* be used for quantification.

So, for example, a selection predicate with a single context set that is $f(\overrightarrow{a}, B) = (B = \varnothing)$ would not be a valid predicate. But often we can write equivalent predicates. For example, in this case, it could be: $f(\overrightarrow{a}, B) = (\forall_c \bot)$.

The recognizer itself can be defined as:

$$r\left(\overrightarrow{A}, \overrightarrow{B}\right) = \{\overrightarrow{a} : \overrightarrow{A} | f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Rightarrow e(\overrightarrow{a})\} \tag{5.4}$$

Equation 5.4 defines how a recognizer operates in terms of its selection and emission functions. For each combination of all inputs, it evaluates the selection predicate for that input and for all computations in all context sets. If the selection predicate evaluates to $\top$, then a computation will exist in the output that is computed by applying the emission to the combination of inputs. Note that the context sets are used to filter combinations of inputs, but they do not affect the actual output once the computations from the inputs have been selected.

Because a recognizer is uniquely defined by its selection predicate and emission function, we often refer to recognizers a pair instead of directly naming the recognizer. So, for example, if f is a selection predicate and e an emission function then $R_{f,e}$ is the recognizer that is defined by that selection predicate and emission function.

The third element in the DAG are oracles. Oracles classify computations into values of $\top$ or $\bot$ and so they are similar to predicates over computations, except that they also output the computation itself along with the classification. Each oracle $\mathcal{O}$ is a function based on a predicate f such that:

$$\begin{aligned}
&\mathcal{O} : \mathbb{C} \rightarrow \mathbb{C} \times \mathbb{B} \\
&f : \mathbb{C} \rightarrow \mathbb{B} \\
&\forall_{a \in \mathbb{C}} \mathcal{O}(a) = (a, f(a))
\end{aligned}$$

In conclusion, a CALL program is a transformation of sets of low-level computations, detected by probes, to high-level classified computations, produced by oracles. The transformation is done through a DAG where each node corresponds to a probe, a recognizer or an oracle. The output corresponding specific input sets of computations is determined by all recognizer's selection predicates and emission functions as well as all oracle's predicates. In the end, by combining the edges of the graph, a single function can be derived that produces sets of classified high-level computations from sets of probed computations.

5.2 Operational Semantics

This section describes how the abstract definitions of recognizer and oracle are implemented in CALL.

In the remainder of this section we will assume that recognizer selection predicates are logical formulas that contain only disjunctions, conjunctions and negations. We do not allow for implication or equivalence operations. This restriction places no constraints in the expressiveness of the language as implication and equivalence are easily rewritten with the other operations.[1]

[1]That is exactly what the CALL compiler does internally when the user provides a predicate with implication or

5.2.1 Recognizer Windows

In Section 5.1 we defined a recognizer as a transformation that uses a vector of i input sets, $\overrightarrow{A} : (\mathbb{PC})^i$, a vector of j context sets, $\overrightarrow{B} : (\mathbb{PC})^j$ a selection predicate $f : \mathbb{C}^i \times (\mathbb{PC})^j \rightarrow \mathbb{B}$ and a emission function $e : \mathbb{C}^i \rightarrow \mathbb{C}$. A recognizer *r* will then produce a single output set $C : \mathbb{PC}$, as:

$$C = r\left(\overrightarrow{A}, \overrightarrow{B}\right) = \{\overrightarrow{a} : \overrightarrow{A} | f\left(\overrightarrow{a}, \overrightarrow{B}\right) \mapsto e\left(\overrightarrow{a}\right)\}$$

While this definition establishes exactly what the output is, it is not usable in practice because it requires the full input and context sets to compute the output and these may not be available. If we were running a system based on past data then all the input sets and context sets may be available, but if we are running a live system, at some point in time some of the events in input and context sets may not yet exist.

In this section we will show that it is possible to define a window that limits how much of the sets we need to see. Intuitively, the window is a sliding window that filters a time-limited subset of computations that the recognizer needs to keep in order to produce the output. By limiting the computations that need to be seen, a recognizer can be operated in a live system not needing to access future computations. This will be discussed in detail later.

So far we have discussed computations without making any assumption on what a computation is or what it contains. To proceed and discuss windows we need to introduce some restrictions on the concept of computation. For the purpose of the semantics, we only need to assume all computations have an *end time*. This is the time at which the computation completed. We will not discuss here delays and how they impact execution. See Section 5.2.8 for a discussion on that topic.

We provide an operator that performs a mapping from $\mathbb{C}$ to $\mathbb{T}$. This operator, the *end-of operator*, when applied to a computation a is written as $\overline{a}$. It can be formally defined as:

$$\forall_{a \in \mathbb{C}} \exists_{t \in \mathbb{T}} t = \overline{a}$$

We extend this definition to vectors of computations. So, for a computation vector $\overrightarrow{a}$ such that $\overrightarrow{a} : \mathbb{C}^i$, we define a time vector $\overrightarrow{t}$ that contains the end time of each computation:

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \forall_{\overrightarrow{t} \in \mathbb{T}^i} \left(\overrightarrow{\overline{a}} = \overrightarrow{t}\right) \Leftrightarrow \left(\forall_{j \in 0..i} \overrightarrow{\overline{a}}[j] = \overrightarrow{t}[j]\right)$$

We also define another operator that applies to vectors of computations and obtains the largest end time of all of the computations in the vector. This operator is named *maximum end time*. For a vector such as $\overrightarrow{a}$, this operator is written as $M(\overrightarrow{a})$ and is defined as:

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} M(\overrightarrow{a}) = \max \overrightarrow{\overline{a}}$$

We define a window as a value drawn from $\mathbb{I}^i$, therefore a multi-dimensional time interval. We define some operations that will be commonly used with windows.

equivalence operators: it converts them into conjunctions, disjunctions and negations. The presence of implications and equivalences would increase the complexity of some analysis that rely on the representation of formulas in disjunctive normal form to detect negated and non-negated terms.

- *Intersection and union*. Given two windows with i dimensions, $\mathcal{W} : \mathbb{I}^i$ and $\mathcal{V} : \mathbb{I}^i$, we can define the intersection and union of the windows as, respectively, $\mathcal{W} \overrightarrow{\cap} \mathcal{V}$ and $\mathcal{W} \overrightarrow{\cup} \mathcal{V}$, as:

$$\forall_{k \in 0..i-1} \left(\mathcal{W} \overrightarrow{\cap} \mathcal{V}\right)[k] = \mathcal{W}[k] \cap_I \mathcal{V}[k]$$
$$\forall_{k \in 0..i-1} \left(\mathcal{W} \overrightarrow{\cup} \mathcal{V}\right)[k] = \mathcal{W}[k] \cup_I \mathcal{V}[k]$$

- *Bounds*. Given a window $\mathcal{W}$, we represent $\mathrm{L}_w(\mathcal{W})$ and $\mathrm{U}_w(\mathcal{W})$ as the window's upper and lower bounds, formally defined as:

$$\forall_{\mathcal{W} \in \mathbb{I}^i} \forall_{\overrightarrow{t} \in \mathbb{T}^i} \left(\mathrm{L}_w(\mathcal{W}) = \overrightarrow{t}\right) \Leftrightarrow \left(\forall_{j \in 0..i-1} \mathrm{L}(\mathcal{W}[j]) = \overrightarrow{t}[j]\right)$$
$$\forall_{\mathcal{W} \in \mathbb{I}^i} \forall_{\overrightarrow{t} \in \mathbb{T}^i} \left(\mathrm{U}_w(\mathcal{W}) = \overrightarrow{t}\right) \Leftrightarrow \left(\forall_{j \in 0..i-1} \mathrm{U}(\mathcal{W}[j]) = \overrightarrow{t}[j]\right)$$

- *Time shift*. A time shift is a translation of the window by a specified amount. Shifting a window $\mathcal{W}$ by t is written as $[\mathcal{W}]^t$. The time shift operator is formally defined as.

$$\forall_{\mathcal{W}, \mathcal{V} \in \mathbb{I}^i} \forall_{t \in \mathbb{T}} \left([\mathcal{W}]^t = \mathcal{V}\right) \Leftrightarrow \left(\forall_{j \in 0..i} [\mathcal{W}[j]]^t = \mathcal{V}[j]\right) \tag{5.5}$$

- Belonging. The belonging operator tests whether a vector of time values belongs to the window. It is a predicate that is verified when all coordinates in the vector are inside the corresponding intervals in the window. For a window $\mathcal{W}$ and time vector $\overrightarrow{t}$ it is written as $\overrightarrow{t} \overrightarrow{\in} \mathcal{W}$. The belonging operator is formally defined as:

$$\forall_{\mathcal{W} \in \mathbb{I}^i} \forall_{\overrightarrow{t} \in \mathbb{T}^i} \overrightarrow{t} \overrightarrow{\in} \mathcal{W} = \left(\forall_{j \in 0..i} \overrightarrow{t}[j] \in \mathcal{W}[j]\right)$$

- Projection. A projection on a window filters a vector of sets of computations by only retaining those whose end fits in the window. The projection of a vector of sets, $\overrightarrow{A}$ into a window $\mathcal{W}$ is written as $\overrightarrow{A} \downarrow_{\mathcal{W}}$ and is formally defined as:

$$\forall_{\mathcal{W} \in \mathbb{I}^i} \forall_{\overrightarrow{A}, \overrightarrow{B} \in (\mathbb{PC})^i} \left(\overrightarrow{A} \downarrow_{\mathcal{W}} = \overrightarrow{B}\right) \Leftrightarrow \left(\forall_{j \in 0..i} \overrightarrow{B}[j] = \{a : \overrightarrow{A}[j] | a \in \mathcal{W}[j]\}\right)$$

We define a windowed recognizer similarly to a recognizer, but constrained to a recognizer window. To do so we need to define a choice function that is used to center the window on one of the input computations. While a recognizer is uniquely defined by its selection predicate and emission function, a windowed recognizer is is uniquely defined by its selection predicate, emission function, window and choice function:

$$R_{f,e,\mathcal{W},p}\left(\overrightarrow{A}, \overrightarrow{B}\right) = \{\overrightarrow{a} : \overrightarrow{A} | \overrightarrow{a} \in [\mathcal{W}]^{\overrightarrow{p(a)}} \wedge f\left(\overrightarrow{a}, \overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overrightarrow{p(a)}}}\right) \Mapsto e\left(\overrightarrow{a}\right)\} \tag{5.6}$$

Since the formula in Equation 5.6 is fundamental, we will break it down. A windowed recognizer is defined by a selection predicate, $f : \mathbb{C}^i \times (\mathbb{PC})^j \rightarrow \mathbb{B}$, an emission function, $e : \mathbb{C}^i \rightarrow \mathbb{C}$ a window $\mathcal{W} : \mathbb{I}^i$ and a choice function, $p : \mathbb{C}^i \rightarrow \mathbb{C}$. A recognizer operates on an vector of input sets, $\overrightarrow{A} : (\mathbb{PC})^i$, and a vector of context sets, $\overrightarrow{B} : (\mathbb{PC})^j$. A vector of input

computations, $\overrightarrow{a} : \mathbb{C}^i$, will only generate an output if (1) $\overrightarrow{a}$ fits in window $\mathcal{W}$ centered on the computation in $\overrightarrow{a}$ selected by the choice function, p, and (2) the selection predicate accepts $\overrightarrow{a}$, but using only the subset of context computations that fit in the window also shifted by the end time of the selected computation. More detailed analysis of this will follow, but it can be seen intuitively that $\mathcal{W}$ limits the amount of past memory a recognizer requires, both for inputs and context. Interestingly, and this will be also discussed in detail later, if $\mathrm{U}_w(\mathcal{W})$ is greater than zero then the recognizer needs future lookahead, but in practice, only for context sets.

The choice function, p, determines which of the computations in $\overrightarrow{a}$ will be used by the recognizer to center the time window. The semantics make no assumptions on the choice function to allow the implementation to pick the most practical one.

We say a window is a valid recognizer window for a predicate, if and only if, the windowed recognizer produces the same output as the recognizer for any input, context, emission function and choice function.

$$\mathrm{valid}\,(\mathcal{W}, f) \Leftrightarrow \left(\forall_{\overrightarrow{A},\overrightarrow{B},e,p} R_{f,e}\left(\overrightarrow{A}, \overrightarrow{B}\right) = R_{f,e,\mathcal{W},p}\left(\overrightarrow{A}, \overrightarrow{B}\right)\right) \tag{5.7}$$

Equation 5.7 along with Theorem 5.2.1 form a critical part of the semantics. Being representable by Equation 5.4 does not imply necessarily that a recognizer is implementable in a real system: according to the definition it needs to know all past and future inputs and computations. Equation 5.6 defines a recognizer that is implementable (with a limited future look-ahead if the window has limited bounds) and Equation 5.7 defines the theoretical condition under which the "conceptual" recognizer can be represented by an "implementable" recognizer. Theorem 5.2.1 establishes concrete tests that must be verified to show that Equation 5.7 is verified.

Theorem 5.2.1. *Let $f : \mathbb{C}^i \times (\mathbb{P}\mathbb{C})^j \to \mathbb{B}$ be a selection predicate and $\mathcal{W} : \mathbb{I}^{i+j}$ be a window. Then,* $\mathrm{valid}\,(\mathcal{W}, f)$ *is equivalent to verifying both Equation 5.8 and Equation 5.9.*

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B}\in(\mathbb{P}\mathbb{C})^j}\forall_{a\in\overrightarrow{a}}\,\overrightarrow{\overrightarrow{a}}\notin [\mathcal{W}]^{\overline{a}} \Rightarrow f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow \bot \tag{5.8}$$

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B},\overrightarrow{C}\in(\mathbb{P}\mathbb{C})^j}\forall_{a\in\overrightarrow{a}}\left(\overrightarrow{B}\downarrow_{[\mathcal{W}]^{\overline{a}}} = \overrightarrow{C}\downarrow_{[\mathcal{W}]^{\overline{a}}}\right) \Rightarrow \left(f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow f\left(\overrightarrow{a}, \overrightarrow{C}\right)\right) \tag{5.9}$$

Proof. We start by proving that a window verifying Equation 5.8 and Equation 5.9 will always be a valid window for a recognizer with selection predicate f.

In the definition of recognizer presented in Equation 5.6 we can split the cases where $\overrightarrow{a}$ is in the window $[\mathcal{W}]^{\overline{a}}$ for every $a \in \overrightarrow{a}$, from the ones it is not. This splits the cases where a choice function could potentially pick a value of a that does not evaluate f unconditionally to $\bot$ regardless of $\overrightarrow{B}$. For ones where the choice function cannot make $\overrightarrow{a}$ fit in the window, the theorem is trivially proved.

For all the cases where the choice function cannot potentially pick a value which fits in the window we know, by Equation 5.8, that $f\left(\overrightarrow{a}, \overrightarrow{B}\right) = \bot$ and so these values will never be selected for emission neither by the windowed recognizer nor by the recognizer.

In the remaining cases there is one a in $\overrightarrow{a}$ such that $\overrightarrow{a}$ fits inside the window when centered on a. In that case we have to prove that the output of the recognizer is the same, regardless of the

context set:

$$f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow f\left(\overrightarrow{a}, \overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overline{a}}}\right)$$

We prove this by contradiction. We suppose that there is at least one a in $\overrightarrow{a}$ and one $\overrightarrow{B}$ for which the equation above is $\bot$. We define $\overrightarrow{C}$ as:

$$\overrightarrow{C} = \overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overline{a}}}$$

In this case, the following equation is trivially $\top$:

$$f\left(\overrightarrow{a}, \overrightarrow{C}\right) \Leftrightarrow f\left(\overrightarrow{a}, \overrightarrow{C} \downarrow_{[\mathcal{W}]^{\overline{a}}}\right)$$

But due to the precondition in Equation 5.9 we know that $f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow f\left(\overrightarrow{a}, \overrightarrow{C}\right)$, meaning there cannot be an a as assumed, proving that, for any $\overrightarrow{a}$, it is accepted by the windowed recognizer if and only if it is accepted by the recognizer.

So far we have proved that a window that verifies Equation 5.8 and Equation 5.9 will be a valid window for the recognizer. Now we have to prove the reverse, that any valid window for the recognizer verifies Equation 5.8 and Equation 5.9.

We start by assuming $\mathcal{W}$ is a valid window for a recognizer $R_{f,e}$, that is, $\text{valid}\left(\mathcal{W}, R_{f,e}\right)$. This mean either window does not validate Equation 5.8, the window does not validate Equation 5.9, or the window doesn't validate both.

We assume $\mathcal{W}$ does *not* validate Equation 5.8 and make no assumptions on whether it validates Equation 5.9. This means that:

$$\exists_{a \in \overrightarrow{a}} \overrightarrow{\overline{a}} \notin [\mathcal{W}]^{\overline{a}} \wedge \left(f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow \top\right)$$

This means the recognizer will accept $\overrightarrow{a}$, even though the windowed recognizer will not. However, if we pick a choose function p that selects a from $\overrightarrow{A}$ ($p\left(\overrightarrow{a}\right) = a$), then $R_{f,e,\mathcal{W},p}$ will not accept $\overrightarrow{a}$ (Equation 5.6). Since we have one windowed recognizer that does not produce the same output as the recognizer, we know from Equation 5.7 that the window is not valid, which contradicts the hypothesis.

So we know that if $\mathcal{W}$ is a valid window for a recognizer $R_{f,e}$, then it must validate Equation 5.8. We will now show it must also validate Equation 5.9. We assume, as before, that $\text{valid}\left(\mathcal{W}, R_{f,e}\right)$, but we assume Equation 5.9 does not hold. This means that there is a $\overrightarrow{a}$, $\overrightarrow{B}$ and a $\overrightarrow{C}$ such that:

$$\left(\overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{W}]^{\overline{a}}}\right) \wedge f\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow \neg f\left(\overrightarrow{a}, \overrightarrow{C}\right)$$

However, if we look at Equation 5.6 and pick a choose function that selects a out of $\overrightarrow{a}$, the value of the selection has to be the same in both, as the windowed recognizer relies on f being applied to only the part of $\overrightarrow{B}$ and $\overrightarrow{C}$ that fit in the window. But since only one of the recognizers will accept $\overrightarrow{a}$ there is at least one windowed recognizer that has a different output meaning, by Equation 5.7, the window is not valid as stated in the hypothesis.

We have shown that if (1) Equation 5.8 and Equation 5.9 are valid for a window then the window is a valid window for the recognizer. We have also shown that if a window is a valid window for a recognizer then both equations also hold proving the theorem. □

Equation 5.8 is essentially stating that if any element in a is outside of the window centered on any element of a, the selection predicate will not accept the set of computations. In other words this means once a computation completes, only computations that have already completed inside the window around that first computation can be selected by the recognizer. Equation 5.8 shows that a recognizer does not need to look more than around the window to identify the sets of computations that can be accepted.

Equation 5.9 is stating that any computations in the context sets outside of the window that contains potentially selectable sets of computations cannot affect the output of selection. These two equations combined show that a recognizer only needs to look in the "immediate surrounding" of the last completed computation. There are two caveats here: one, the theorem does not place any bounds of the window so a specific recognizer may require an infinitely large window; two, while we can get around requiring future computations for the $\overrightarrow{a}$ set by picking a choose function that selects the last computation in $\overrightarrow{a}$ to center the window, the selection of computations for context *may* include future values. This will be discussed in Section 5.2.8.

Additionally, while theorem 5.2.1 states the conditions a valid window must obey to allow the use if a windowed recognizer instead of a recognizer, it doesn't state *how* to compute such window. That is the topic of the following sections.

5.2.2 Disjunctive Normal Form

We say a three-valued propositional logic formula is in *disjunctive normal form* (or DNF) if it is a disjunction of one or more formulas, which we call constraints, and each constraint is a conjunction of one or more terms, each term being a (possibly negated) variable. This is the same definition of DNF commonly used for propositional logic formulas. Note that a disjunction of one constraint is simply that constraint and conjunction of one term is simply that term. A term is named *positive* if it is not negated and *negative* if it is negated.

Terms should not be mistaken with variables. For example, consider for formula in Equation 5.10. The formula has two variables, q and r, two constraints, q and $r \vee \neg q$ and three terms, q, r and $\neg q$, the first two positive and the last one negative. The same term may appear in multiple constraints and may even appear multiple times in a constraint, although the last case is easily simplifiable as $(q \vee q) \Leftrightarrow q$.

$$q \vee (r \wedge \neg q) \tag{5.10}$$

Formulas are not necessarily written in DNF. Any formula can be defined as a tree with the three operators (negation, conjunction and disjunction) as non-leaf nodes and variables as leaf nodes. This defines a syntax tree for the formula. For example we can write $(q \vee \neg r) \wedge s$ as: $\wedge(\vee(q, \neg(r)), s)$.

We define the $\mathscr{C}$ operator that transforms a propositional logic formula into an equivalent one in DNF format. This operator transforms a formula according to the following algorithm:

1. For every negated conjunction or disjunction, apply DeMorgan's law. Repeat until there are no more negated conjunctions or disjunctions.
2. If there are conjunctions of disjunctions, apply distributivity to the first such occurrence in the tree.

3. Merge all conjunctions of conjunctions and disjunctions of disjunctions.
4. Repeat from Step 2 until there are no conjunctions of disjunctions.
5. Remove all double negations.

Theorem 5.2.2. *When applying $\mathscr{C}$ to any propositional logic formula:*

1. *The algorithm terminates.*
2. *The resulting formula is equivalent to the original one (produces the same output for the same input).*
3. *The resulting formula is in DNF format.*

Proof. We start by proving that the algorithm terminates. All steps individually clearly terminate: (a) DeMorgan's law pushes negations down the syntax tree, but doesn't change the height of the tree so it has to terminate at some point. (b) Applying distributivity and merging conjunctions and disjunctions will push conjunctions down the tree, but won't increase the tree's height so they are guaranteed to terminate. (c) Removing double negations removes the number of negation nodes so we will eventually run out of nodes to remove.

Equivalence is trivial since all transformations done in the tree preserve equivalence.

Lastly, the algorithm terminates when there are no negations over conjunctions or disjunctions, only over individual terms (this is guaranteed by the first step), there are no conjunctions of disjunctions, only disjunctions of conjunctions. This means the disjunctions are at the top of the tree and the conjunctions at the bottom. Because all disjunctions are merged, this implies that there will only be one disjunction of conjunctions. This is, by definition, a DNF formula. □

Theorem 5.2.3. *Let $f : \mathbb{B} \to \mathbb{B}$ be a boolean DNF formula that evaluates a single boolean variable that appears in either only positive or negative terms. Then one of the following must be $\top$:*

- $f(q) \Leftrightarrow \top$
- $f(q) \Leftrightarrow \bot$
- $f(q) \Leftrightarrow q$ *(if the term appears non-negated in the DNF formula)*
- $f(q) \Leftrightarrow \neg q$ *(if the term appears negated in the DNF formula)*

Proof. If any of the restrictions of f that is not the one containing q evaluates to $\top$, then regardless of q, f will evaluate to $\top$. Otherwise, all restrictions of f that are not one containing q must evaluate to $\bot$ and do not influence the result.

In the latter case if any of the terms of the restriction of f that contains q evaluates to $\bot$ then f evaluates always to $\bot$. Otherwise, none of the terms of the restriction of f containing q evaluate to $\bot$ and none of them has any influence on the result.

In the latter case f evaluates to the term which will either be q is the term is non-negated or $\neg q$ if the term is negated. □

5.2.3 Computing Windows of Non-Quantified Formulas

Theorem 5.2.1 shows that if we have a valid window for a recognizer, then we can use it to make a recognizer that doesn't need all the computations in the input and context sets, but only those that fit in the window. In this section we show how such a window can be computed for selection

predicates that do not have quantifiers and, therefore, do not depend on any context. In these cases, due to the absence of context in the selection predicate, the only equation relevant for computing a window is Equation 5.8.

To compute the window for a predicate, we define a *timed predicate*. A timed predicate is a propositional logic predicate that is the result of a syntax tree transformation of another predicate. The formula of the source predicate must be a conjunction of terms. The transformation is done by:

- Removing any terms in the conjunction except those that only involve constants and *end of* operators.
- Replace every *end of* with a timed variable.

If f is a predicate, $T[f]$ is the equivalent timed predicate. For example, if f is defined as below, then g, shown below, is the result of $T[f]$.

$$f(a,b) = \left(\overline{a} <_t \overline{b}\right) \wedge \left(\overline{b} <_t \overline{a} +_t 10s\right) \wedge (\mathtt{id}(a) = \mathtt{id}(b)) \tag{5.11}$$

$$g(t,u) = T[f] = (t <_t u) \wedge (u <_t t +_t 10s) \tag{5.12}$$

Timed predicates allow us to discuss predicates moving from the $\mathbb{C}^i$ space into the $\mathbb{T}^i$ space. Timifying is useful because because it simplifies the formulas by removing all information not pertaining to time. It transforms formulas into pure time constraints, while still being usable to derive conclusions about the original predicate (see Theorem 5.2.4).

Theorem 5.2.4. *Let $f : \mathbb{C}^i \to \mathbb{B}$ be a propositional logic predicate whose formula is a conjunction of terms over i computations. Let $g : \mathbb{T}^i \to \mathbb{B}$ be $g = T[f]$. Then:*

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \neg g\left(\overline{\overrightarrow{a}}\right) \Rightarrow \neg f\left(\overrightarrow{a}\right)$$

Proof. If g evaluates to $\perp$ then it means one of the terms in the conjunction evaluates to $\perp$. This term must also exist in f by construction so f must also evaluate to $\perp$. □

Theorem 5.2.5. *Let $f : \mathbb{C}^i \to \mathbb{B}$ be a predicate and let $\mathcal{W}$ be a window in the $\mathbb{I}^i$ space such that*

$$\forall_{\overrightarrow{t} \in \mathbb{T}^i} \left(\forall_{t \in \overrightarrow{t}} \overrightarrow{t} \notin [\mathcal{W}]^t\right) \Rightarrow T[f]\left(\overrightarrow{t}\right) = \perp$$

Then Equation 5.8 is valid for f and $\mathcal{W}$.

Proof. Let $\overrightarrow{a} : \mathbb{C}^i$ be any set of i computations. Let $\overrightarrow{t} = \overline{\overrightarrow{a}}$. The left-hand side of the equation can be rewritten to match the one in Equation 5.8. A direct replace yields:

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \left(\forall_{a \in \overrightarrow{a}} \overline{\overrightarrow{a}} \notin [\mathcal{W}]^{\overline{a}}\right) \Rightarrow T[f]\left(\overline{\overrightarrow{a}}\right) = \perp$$

But because $\left(\forall_{a \in \overrightarrow{a}}\right) \Rightarrow \left(\exists_{a \in \overrightarrow{a}}\right)$, we can write:

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \left(\exists_{a \in \overrightarrow{a}} \overline{\overrightarrow{a}} \notin [\mathcal{W}]^{\overline{a}}\right) \Rightarrow T[f]\left(\overline{\overrightarrow{a}}\right) = \perp$$

Due to Theorem 5.2.4, we know that if $T[f]\left(\overline{\overrightarrow{a}}\right) = \perp$, then $f\left(\overrightarrow{a}\right) = \perp$. □

Theorem 5.2.6. *Let $f : \mathbb{C}^i \to \mathbb{B}$ be a selection predicate over i inputs and with no context and let $\mathcal{W}$ be a valid window for f. Let $g : \mathbb{C}^i \to \mathbb{B}$ be another selection predicate over i inputs and with no context and let $\mathcal{V}$ be a valid window for g. Then $\mathcal{W} \overrightarrow{\cup} \mathcal{V}$ is a valid window for the selection predicate $f \vee g$.*

Proof. Let $\overrightarrow{a} : \mathbb{C}^i$. If $\overrightarrow{a \notin} (\mathcal{W} \overrightarrow{\cup} \mathcal{V})$, then $\left(\overrightarrow{a \notin} \mathcal{W}\right) \wedge \left(\overrightarrow{a \notin} \mathcal{V}\right)$. This means $f(\overrightarrow{a}) = \perp$ and $g(\overrightarrow{a}) = \perp$, which is equivalent to saying $(f \vee g)(\overrightarrow{a}) = \perp$, which, given that there is no context, shows the window is valid. □

With the timing predicates, Theorem 5.2.5 and Theorem 5.2.6 we have further simplified the problem of determining a window for a recognizer. If we have a selection predicate in DNF then we can compute a window for each timified restriction and use Theorem 5.2.6 to compute a window for the predicate. This means all that is missing is a way to compute a window for a single restriction that only contains constants and time variables.

CALL has strong limitations on how timing predicates can be defined. The details are provided in Appendix B, but the relevant rules are:

- Time values can only be obtained by extracting the end time of computations.
- Two time values can only be compared to each other or subtracted and the result of subtraction of time values is a (possibly negative) duration.
- Durations can be added or subtracted to time values.
- Durations can be added and subtracted and can be multiplied by constants.
- Durations can be specified as literals in the predicate.

These rules have two important implications: All terms in timified conjunctions are linear equations (or inequalities) and the sum of all coefficients of all variables in every such linear equation is zero (see Theorem 5.2.7). The fact that the sum of all coefficients is zero means that if we translate all computations by the same time quantity then the new set of computations will fit in the window only if the initial set did. This is an intuitive result as recognizers are matching only computations based on their relative times. A formal proof is provided in Theorem 5.2.8.

Theorem 5.2.7. *Let $f : \mathbb{C}^i \to \mathbb{B}$ be a term in a* CALL *selection predicate without quantifiers that is neither a conjunction nor disjunction of terms. Then $T[f]$ is a linear equation (or inequality) in the end time of computations. The sum of all time coefficients in this linear equation is zero.*

Proof. Trivial from the structure of CALL predicates since time values must always appear in pairs with one being positive and another negative since the only operations allowed are comparison and subtraction. □

Theorem 5.2.8. *Let $f : \mathbb{C}^i \to \mathbb{B}$ be a* CALL *predicate without quantifiers. Let j any number in the $0..i-1$ range. Let $\mathcal{W}$ be a window such that:*

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \left(\overline{\overrightarrow{a}[j]} = 0s\right) \wedge \overrightarrow{\overrightarrow{a} \notin} \mathcal{W} \Rightarrow (f(\overrightarrow{a}) = \perp) \tag{5.13}$$

Then:

$$\forall_{\overrightarrow{a} \in \mathbb{C}^i} \overrightarrow{\overrightarrow{a} \notin} [\mathcal{W}]^{\overrightarrow{\overrightarrow{a}[j]}} \Rightarrow (f(\overrightarrow{a}) = \perp) \tag{5.14}$$

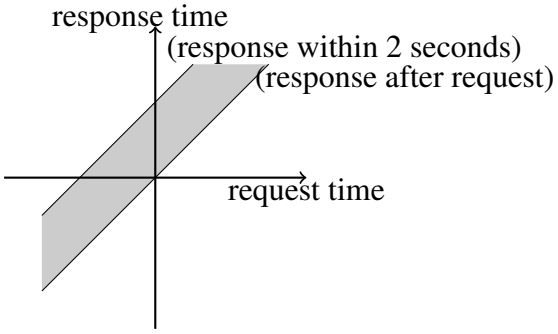


Figure 5.2: Example showing the acceptable times for a response/request.

Proof. Let's assume there is a vector of computations, $\overrightarrow{b} : \mathbb{C}^i$, such that $\overrightarrow{\overrightarrow{b}\overrightarrow{b}} \notin [\mathcal{W}]^{\overrightarrow{b}[j]}$, but for which $f\left(\overrightarrow{b}\right) \neq \perp$. Then let $\overrightarrow{c} : \mathbb{C}^i$ be a set of computations that are exactly the same as $\overrightarrow{b}$, but whose end times are all shifted by $-\overrightarrow{b}[j]$. Then, due to the precondition of the theorem, we would have $f(\overrightarrow{c}) = \perp$. However, due to Theorem 5.2.7, every single term in f has coefficients for time adding up to zero and so it is not possible for $f\left(\overrightarrow{b}\right) \neq f(\overrightarrow{c})$, proving the theorem. □

Theorem 5.2.8 makes use of the translation properties of the selection predicate to state a useful result: if we select one input set and if we have a window such that any selection of other input computations that falls outside of the window is rejected by the selection predicate, then this window can be used to reject any combination of computations as long as it is centered on the computation of the selected dimension.

For example, consider a selection predicate that accepts a response coming up to 2 seconds after a request. If the request comes at $0s$ then the window $([0s, 0s],]0s, 2s[)$ would reject only invalid combinations of request/response. For a request coming at $5s$, then the window $([5s, 5s],]5s, 7s[)$ would do the same. This can be depicted graphically in Figure 5.2. The area in shade is the area where a request/response time is accepted by the window.

In general, due to Theorem 5.2.7, the areas of validity are always representable by "sliding" along the unity vector. This allows us to fix one dimension and compute the window centered on that computation, which is essentially what Theorem 5.2.8 is doing (see Figure 5.3).

For the simple request/response case presented above we know, for any acceptable case, that the response will come after the response. Since the recognizer will center the window around the latest computation (Equation 5.8), we only need to center the window around the response. This conforms to the intuition that if a response must come within two seconds of a request, the recognizer only need to keep track of the requests of the past two seconds. But, in the general case, the order by which the computations is not necessarily fixed. Additionally, for computation in context sets, which we are not dealing with in this section, but will in a later section, we cannot center the window in the context computation, even if it is the last one. So we have to look at the windows centering on all computations. This lead us to Theorem 5.2.9.

Theorem 5.2.9. *Let i be a number of dimensions. Let $f : \mathbb{C}^i \to \mathbb{B}$ be a predicate over i*

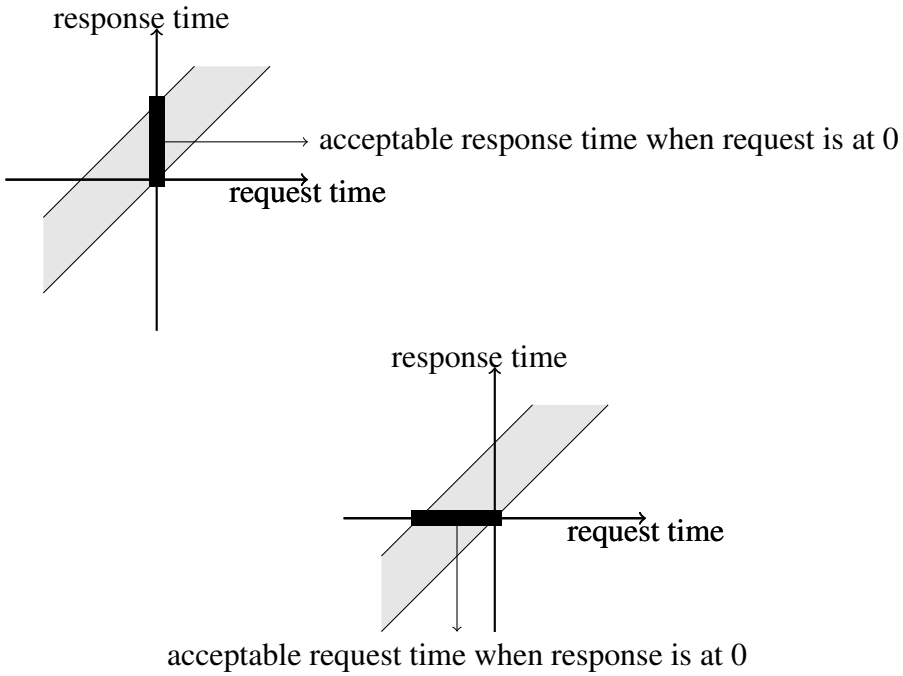


Figure 5.3: Defining windows by centering around one computation at zero.

computations. Let $\mathcal{W}_0$ up to $\mathcal{W}_{i-1}$ be a set of i windows and $\mathcal{V}$ be a window that is the union of all $\mathcal{W}$ windows. Then:

$$\left(\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{j\in 0..i-1}\overrightarrow{a}\notin [\mathcal{W}_j]^{\overrightarrow{a}[j]} \Rightarrow f(\overrightarrow{a}) = \bot\right) \Rightarrow \left(\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{a\in\overrightarrow{a}}\overrightarrow{a}\notin \left[\bigcup_j \mathcal{W}_j\right]^{a} \Rightarrow f(\overrightarrow{a}) = \bot\right)$$

Proof. If $\overrightarrow{a}$ does not fit in $\left[\bigcup_j \mathcal{W}_j\right]^{a}$, then it doesn't fit in $[\mathcal{W}_j]^{\overrightarrow{a}[j]}$. □

Theorem 5.2.9 establishes that if we compute a window in the way defined by Theorem 5.2.8 for each dimension, independently, we can union all the windows to find a window for Equation 5.8. This is exemplified in Figure 5.4.

In the example of request/response we've been using, given a request computation at some time, to identify the range of times allowed for the responses, we need to shift the window by the time of the request computation. Shifting is done by applying the same amount to all dimensions, so it translates diagonally in the diagram (see Equation 5.5 and Section 5.1.1).

Theorem 5.2.7, Theorem 5.2.8 and Theorem 5.2.9 allow us to compute a window for a formula by iterating over all variables, setting the end time of the computation to $0s$ and computing a window for the remaining variables. Because these conjunctions of terms are always linear, any space restrained by them is convex. We can simply enumerate all vertices to compute their windows. The window will be comprised of all the minimum and maximum values of all vertices. Then, we can compute the union of all the windows and obtain a window for the full non-quantified formula.

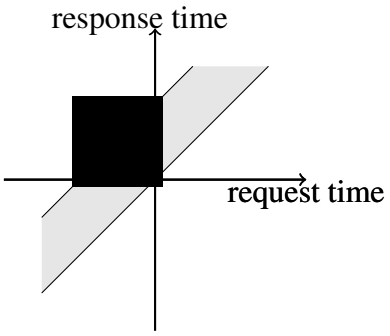


Figure 5.4: Unioning the windows from Figure 5.3.

The results presented in this section allow us to pick a first order logic predicate without quantifiers and compute a window such that events outside of the window are never accepted by the predicate and, thus, never correspond to outputs of the recognizer. To do this, we follow the steps described below:

1. Transform the predicate into DNF.
2. Compute the corresponding timed predicate.
3. For each restriction in the DNF formula:
   (a) For each dimension, compute a window for the restriction centered on a computation at time $0s$ in the chosen dimension.
   (b) Union all the windows for each dimension to compute a window for the restriction.
4. Union all the windows for all the restrictions to compute a window for the predicate.

In the next section we will discuss how to extend the algorithm described in this section to selection predicates with quantifiers.

5.2.4 Computing Windows for Formulas With Quantifiers

In this section we will show how to extend the results from Section 5.2.3 to include formulas with quantifiers. The mechanism to do this is described below and supported by Theorem 5.2.10, presented after the description.

Given a formula with quantifiers we incrementally build the window by following these steps:

1. Split the formula into sub-formulas by quantifiers. Quantifiers in formulas are replaced by a boolean variable and the quantified expression becomes a formula itself. For example, consider the following formula for $f\,(a, b, C, D)$:

   $$\overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \left(\forall_{c \in C} \overline{c} <_t \overline{a} \vee \overline{c} >_t \overline{b}\right) \wedge \exists_{d \in D} \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s \wedge \exists_{c \in C} \overline{c} >_t \overline{d} \wedge \overline{c} <_t \overline{a}$$

   This formula states that two events a and b must come in order, separated by at most 10 seconds. There are three additional restrictions: there must be no c between them (so all c computations must occur either before a or b), there must be at least one computation d occurring in the the 5 seconds that preceed a and one c after d, but before a. This can be depicted graphically as in Figure 5.5.

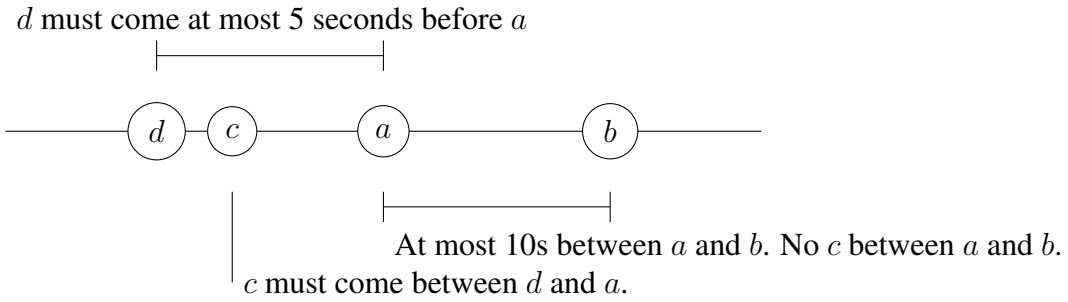


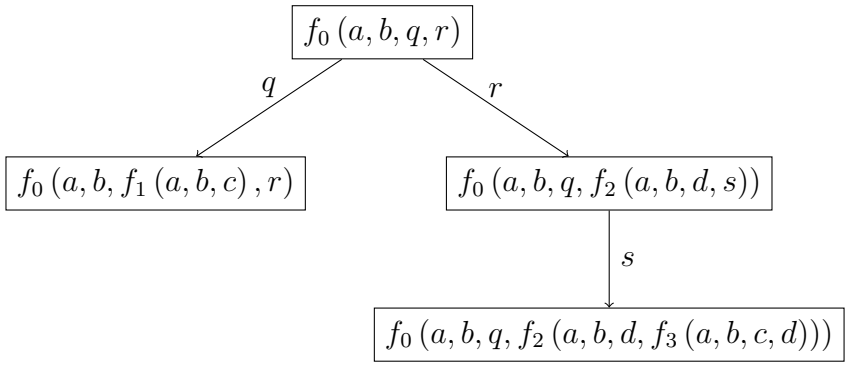
Figure 5.5: Restrictions of computations for example to compute a window over a quantified formula.

This formula will be broken down in the sub-formulas below (f_0 to f_3). The appearance of some negated terms is due to Theorem 5.2.10, which tells us terms in universal quantifiers should be negated.

$$
\begin{aligned}
f_0\left(a,b,q,r\right) &= \overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge q \wedge r \\
f_1\left(a,b,c\right) &= \neg\left(\overline{c} <_t \overline{a} \vee \overline{c} >_t \overline{b}\right) \\
f_2\left(a,b,d,s\right) &= \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s \wedge s \\
f_3\left(a,b,c,d\right) &= \overline{c} >_t \overline{d} \wedge \overline{c} <_t \overline{a}
\end{aligned}
$$

2. Build a tree where each node corresponds to a predicate. The root node is f_0. Each edge going down is associated with a boolean variable in the predicate and the subsequent node is obtained by replacing the boolean variable with the equivalent predicate. Then one edge is added per boolean variable in the equivalent predicate (not the full formula) and so on.

   In the example above, the tree would look something like:



3. Compute a window for each predicate in the tree that is valid for all combinations of all boolean values that exist in the predicate. With Theorem 5.2.11, this means computing a window removing all boolean variables.

   In the example case we have to compute windows for:

   (a) $f_0\left(a,b\right) = \overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s$.

$\overline{a} = 0$	$0 <_t \overline{b} \wedge \overline{b} <_t 0 +_t 10s.$
	$([0,0],]0,10[)$
$\overline{b} = 0$	$\overline{a} <_t 0 \wedge 0 <_t \overline{a} +_t 10s.$
	$(]-10,0[,[0,0])$
$\mathcal{W} = (]-10,0],[0,10[)$	

(b) $f_0(a,b,f_1(a,b,c)) = \overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \neg\left(\overline{c} <_t \overline{a} \vee \overline{c} >_t \overline{b}\right)$

$\overline{a} = 0$	$0 <_t \overline{b} \wedge \overline{b} <_t 0 +_t 10s \wedge \neg\left(\overline{c} <_t 0 \vee \overline{c} >_t \overline{b}\right)$
	$([0,0],]0,10[,[0,10[)$
$\overline{b} = 0$	$\overline{a} <_t 0 \wedge 0 <_t \overline{a} +_t 10s \wedge \neg(\overline{c} <_t \overline{a} \vee \overline{c} >_t 0)$
	$(]-10,0[,[0,0],]-10,0])$
$\overline{c} = 0$	$\overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \neg\left(0 <_t \overline{a} \vee 0 >_t \overline{b}\right)$
	$(]-10,0[,]0,10[,[0,0])$
$\mathcal{W}_q = (]-10,0],[0,10[,]-10,10[)$	

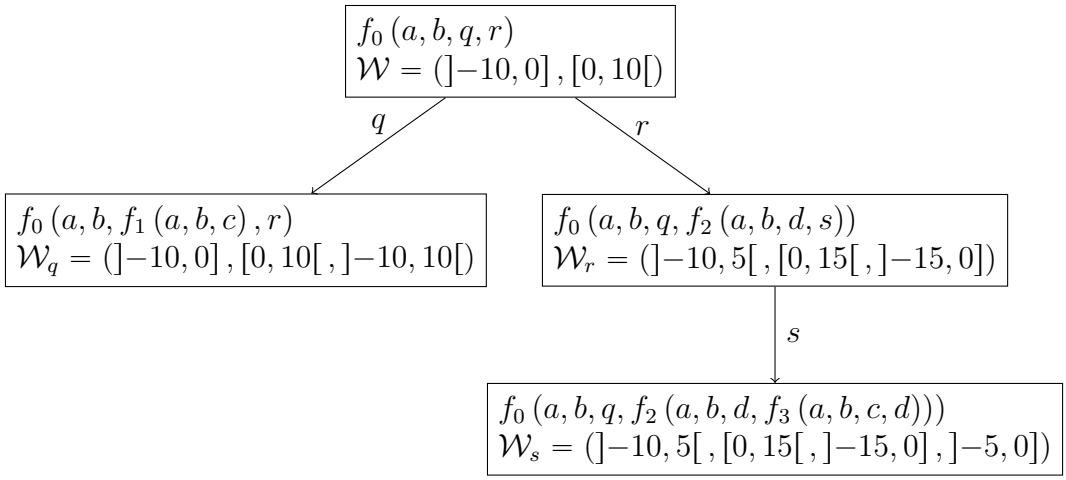
(c) $f_0(a,b,f_2(a,b,d)) = \overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s$

$\overline{a} = 0$	$0 <_t \overline{b} \wedge \overline{b} <_t 0 +_t 10s \wedge \overline{d} <_t 0 \wedge \overline{d} >_t 0 -_t 5s$
	$([0,0],]0,10[,]-5,0[)$
$\overline{b} = 0$	$\overline{a} <_t 0 \wedge 0 <_t \overline{a} +_t 10s \wedge \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s$
	$(]-10,0[,[0,0],]-15,-10[)$
$\overline{d} = 0$	$\overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge 0 <_t \overline{a} \wedge 0 >_t a -_t 5s$
	$(]0,5[,]0,15[,[0,0])$
$\mathcal{W}_r = (]-10,5[,[0,15[,]-15,0])$	

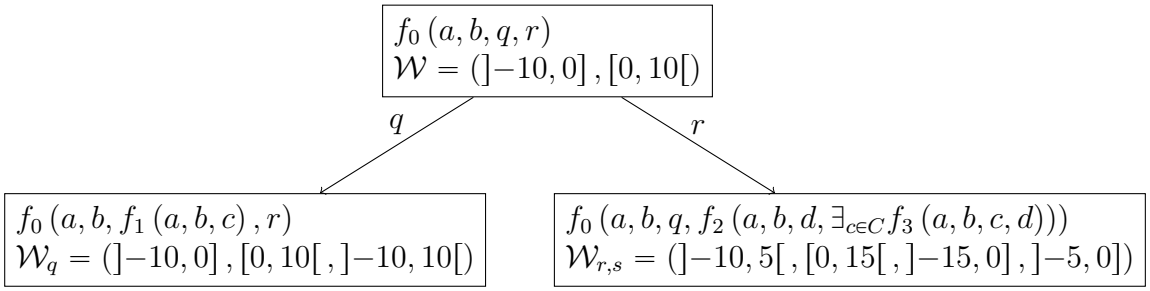
(d) $f_0(a,b,f_2(a,b,d,f_3(a,b,c,d))) = \overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s \wedge \overline{c} >_t \overline{d} \wedge \overline{c} <_t \overline{a}$

$\overline{a} = 0$	$0 <_t \overline{b} \wedge \overline{b} <_t 0 +_t 10s \wedge \overline{d} <_t 0 \wedge \overline{d} >_t 0 -_t 5s \wedge \overline{c} >_t \overline{d} \wedge \overline{c} <_t \overline{a}$
	$([0,0],]0,10[,]-5,0[,]-5,0[)$
$\overline{b} = 0$	$\overline{a} <_t 0 \wedge 0 <_t \overline{a} +_t 10s \wedge \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s \wedge \overline{c} >_t \overline{d} \wedge \overline{c} <_t \overline{a}$
	$(]-10,0[,[0,0],]-15,-5[,]-5,0[)$
$\overline{c} = 0$	$\overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge \overline{d} <_t \overline{a} \wedge \overline{d} >_t a -_t 5s \wedge 0 >_t \overline{d} \wedge 0 <_t \overline{a}$
	$(]0,5[,]0,15[,[0,0],]-5,0[)$
$\overline{d} = 0$	$\overline{a} <_t \overline{b} \wedge \overline{b} <_t \overline{a} +_t 10s \wedge 0 <_t \overline{a} \wedge 0 >_t a -_t 5s \wedge \overline{c} >_t 0 \wedge \overline{c} <_t \overline{a}$
	$(]0,5[,]0,15[,]0,5[,[0,0])$
$\mathcal{W}_s = (]-10,5[,[0,15[,]-15,0],]-5,0])$	

In the example case we end up with:



4. For each node that only contains sub nodes that are leaf nodes, remove the leaf nodes, add the quantifier and recompute the window using Theorem 5.2.10.
5. Repeat until there is only the root node. After the first iteration the result would be:



   After the second iteration the result would be:

$$f_0(a,b,\forall_{c\in C} f_1(a,b,c),\exists_{d\in D} f_2(a,b,d,\exists_{c\in E} f_3(a,b,c,d)))$$
$$\mathcal{W}_{q,r,s} = (]-10,5[,[0,15[,]-10,10[,]-15,0],]-5,0])$$

6. For any quantifiers over sets that are the same in the original predicate but are different sets in this predicate, "merge" the sets using Theorem 5.2.13. In the example the end result would be:

$$f_0(a,b,\forall_{c\in C} f_1(a,b,c),\exists_{d\in D} f_2(a,b,d,\exists_{c\in C} f_3(a,b,c,d)))$$
$$\mathcal{V} = (]-10,5[,[0,15[,]-10,10[,]-5,0])$$

7. This final predicate is the same as the original predicate and the window is a valid window for the predicate.

At the end of this process we have a valid window for the predicate. In the remainder of this section we will present the theorems referred to by the algorithm.

Theorem 5.2.10. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \times \mathbb{B} \times \mathbb{B}^k \to \mathbb{B}$ be a predicate that evaluates i computations in the context of j sets of computations and depends on $k+1$ boolean values with the first of the boolean variable appearing only once as a non-negated (†) term in the CNF representation of the function. Let $\mathcal{W} : \mathbb{I}^{i+j}$ be a valid window for f for each combination of boolean values, that is, $\forall_{\overrightarrow{q}\in\mathbb{B}^{j+1}} \text{valid}\left(\mathcal{W}, \lambda_{\overrightarrow{a}:\mathbb{C}^i,B:(\mathbb{PC})^j}.f\left(\overrightarrow{a},\overrightarrow{B},\overrightarrow{q}\right)\right)$.*

Let $g : \mathbb{C}^{i+1} \times (\mathbb{PC})^l \to \mathbb{B}$ be a predicate that evaluates $i+1$ computations in the context of l sets of computations. Let $\mathcal{V} : \mathbb{I}^{i+j+l+1}$ be a valid window for $\lambda_{\overrightarrow{a}:\mathbb{C}^i,\overrightarrow{B}:(\mathbb{PC})^j,c:\mathbb{C},\overrightarrow{C}:(\mathbb{PC})^l}.f\left(\overrightarrow{a},\overrightarrow{B},g\left(\overrightarrow{a},c,\overrightarrow{C}\right),\overrightarrow{q}\right)$ for any $\overrightarrow{q} : \mathbb{B}^k$.

Let $\mathcal{U} : \mathbb{I}^{i+j+l+1}$ be a window defined as $\mathcal{U}[m] = \mathcal{W}[m] \cup \mathcal{V}[m]$, for every $m \in 0..i+j-1$, and $\mathcal{U}[m] = \mathcal{V}[m]$, for every $l \in i+j..i+j+l$.

Then, for any $\overrightarrow{q} : \mathbb{B}^k$, $\mathcal{U}$ is a valid window for both Equation 5.15 and Equation 5.16.

$$\lambda_{\overrightarrow{a}:\mathbb{C}^i,\overrightarrow{B}:(\mathbb{PC})^{j+l+1}}.f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\forall_{c\in\overrightarrow{B}_{[j]}}\neg g\left(\overrightarrow{a},c,\overrightarrow{B}_{j+1..j+l}\right),\overrightarrow{q}\right) \tag{5.15}$$

$$\lambda_{\overrightarrow{a}:\mathbb{C}^i,\overrightarrow{B}:(\mathbb{PC})^{j+l+1}}.f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\exists_{c\in\overrightarrow{B}_{[j]}}g\left(\overrightarrow{a},c,\overrightarrow{B}_{j+1..j+l}\right),\overrightarrow{q}\right) \tag{5.16}$$

(†) if the term appears negated, the theorem can applied by negating the quantifiers.

Proof. To prove the result we need to show that Equations 5.8 and 5.9 are valid. If we do, then from Theorem 5.2.1, we know $\mathcal{U}$ is a valid window for both formulas.

We start by proving Equation 5.8. This means we need to prove the following two formulas:

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B}\in(\mathbb{PC})^j}\forall_{\overrightarrow{C}\in(\mathbb{PC})^{j+1}}\forall_{a\in\overrightarrow{a}}\overrightarrow{\overrightarrow{a}\notin}\,[\mathcal{U}]^{\overline{a}} \Rightarrow f\left(\overrightarrow{a},\overrightarrow{B},\forall_{c\in\overrightarrow{C}_{[0]}}\neg g\left(\overrightarrow{a},c,\overrightarrow{C}_{-0}\right),\overrightarrow{q}\right) = \bot$$

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B}\in(\mathbb{PC})^j}\forall_{\overrightarrow{C}\in(\mathbb{PC})^{j+1}}\forall_{a\in\overrightarrow{a}}\overrightarrow{\overrightarrow{a}\notin}\,[\mathcal{U}]^{\overline{a}} \Rightarrow f\left(\overrightarrow{a},\overrightarrow{B},\exists_{c\in\overrightarrow{C}_{[0]}}g\left(\overrightarrow{a},c,\overrightarrow{C}_{-0}\right),\overrightarrow{q}\right) = \bot$$

Both these equations are proved in the same way. Regardless of the value of $a \in \overrightarrow{a}$, if $\overrightarrow{\overrightarrow{a}\notin}\,[\mathcal{U}]^{\overline{a}}$, then $\overrightarrow{\overrightarrow{a}\notin}\,[\mathcal{W}]^{\overline{a}}$ because for m in $0..i-1$, $\mathcal{U}[m] = \mathcal{W}[m] \cup \mathcal{V}[m]$. We know $\mathcal{W}$ is a valid window for $f\left(\overrightarrow{a},\overrightarrow{B},r,\overrightarrow{q}\right)$ for any $r : \mathbb{B}$ and any $\overrightarrow{q} : \mathbb{B}^k$, so we know that f must evaluate to $\bot$.

This proves Equation 5.8, but we also have to prove Equation 5.9. Tha requires proving that, regardless of the value of $\overrightarrow{q} : \mathbb{B}^j$:

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B}\in(\mathbb{PC})^{j+l+1}}\forall_{\overrightarrow{C}\in(\mathbb{PC})^{j+l+1}}\forall_{a\in\overrightarrow{a}}\left(\overrightarrow{B}\downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C}\downarrow_{[\mathcal{U}]^{\overline{a}}}\right)$$
$$\Rightarrow \left(f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\forall_{c\in\overrightarrow{B}_{[j]}}\neg g\left(\overrightarrow{a},c,\overrightarrow{B}_{j+1..j+l}\right),\overrightarrow{q}\right) = f\left(\overrightarrow{a},\overrightarrow{C}_{0..j-1},\forall_{c\in\overrightarrow{C}_{[j]}}\neg g\left(\overrightarrow{a},c,\overrightarrow{C}_{j+1..j+l}\right),\overrightarrow{q}\right)\right)$$

$$\forall_{\overrightarrow{a}\in\mathbb{C}^i}\forall_{\overrightarrow{B}\in(\mathbb{PC})^{j+l+1}}\forall_{\overrightarrow{C}\in(\mathbb{PC})^{j+l+1}}\forall_{a\in\overrightarrow{a}}\left(\overrightarrow{B}\downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C}\downarrow_{[\mathcal{U}]^{\overline{a}}}\right)$$
$$\Rightarrow \left(f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\exists_{c\in\overrightarrow{B}_{[j]}}g\left(\overrightarrow{a},c,\overrightarrow{B}_{j+1..j+l}\right),\overrightarrow{q}\right) = f\left(\overrightarrow{a},\overrightarrow{C}_{0..j-1},\exists_{c\in\overrightarrow{C}_{[j]}}g\left(\overrightarrow{a},c,\overrightarrow{C}_{j+1..j+l}\right),\overrightarrow{q}\right)\right)$$

Given the $\overrightarrow{B}\downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C}\downarrow_{[\mathcal{U}]^{\overline{a}}}$, we know that $f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\top,\overrightarrow{q}\right) = f\left(\overrightarrow{a},\overrightarrow{C}_{0..j-1},\top,\overrightarrow{q}\right)$ and $f\left(\overrightarrow{a},\overrightarrow{B}_{0..j-1},\bot,\overrightarrow{q}\right) = f\left(\overrightarrow{a},\overrightarrow{C}_{0..j-1},\bot,\overrightarrow{q}\right)$, due to the theorem's preconditions as $\mathcal{W}$ is contained in $\mathcal{U}$ and $\mathcal{W}$ is a valid window for f, regardless of the value of the $k+1$ boolean variables, which includes $\overrightarrow{q}$ and the isolated boolean variable replacing the quantifier.

What we don't know is whether $f\left(\overrightarrow{a}, \overrightarrow{B}_{0..j-1}, \top, \overrightarrow{q}\right) = f\left(\overrightarrow{a}, \overrightarrow{C}_{0..j-1}, \bot, \overrightarrow{q}\right)$. For the values of $\overrightarrow{a}$, $\overrightarrow{B}$, $\overrightarrow{C}$ and a for which this last equality holds, Equation 5.9 is verified. In these cases, the value taken by g is irrelevant.

For the remaining combinations of $\overrightarrow{a}$, $\overrightarrow{B}$ $\overrightarrow{C}$ and a, we can make use of Theorem 5.2.3 and just keep the third parameter of f:

$$\left(\overrightarrow{B} \downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{U}]^{\overline{a}}}\right) \Rightarrow \left(\left(\forall_{c \in \overrightarrow{B}_{[j]}} \neg g\left(\overrightarrow{a}, c, \overrightarrow{B}_{j+1..j+l}\right)\right) = \forall_{c \in \overrightarrow{C}_{[j]}} \neg g\left(\overrightarrow{a}, c, \overrightarrow{C}_{j+1..j+l}\right)\right)$$
$$\left(\overrightarrow{B} \downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{U}]^{\overline{a}}}\right) \Rightarrow \left(\left(\exists_{c \in \overrightarrow{B}_{[j]}} g\left(\overrightarrow{a}, c, \overrightarrow{B}_{j+1..j+l}\right)\right) = \exists_{c \in \overrightarrow{C}_{[j]}} g\left(\overrightarrow{a}, c, \overrightarrow{C}_{j+1..j+l}\right)\right)$$

These two formulas are actually equivalent since the negation can be moved out of the universal quantifier, transforming the quantifier into an existential quantifier. We only have to prove the second formula:

$$\left(\overrightarrow{B} \downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{U}]^{\overline{a}}}\right) \Rightarrow \left(\left(\exists_{c \in \overrightarrow{B}_{[j]}} g\left(\overrightarrow{a}, c, \overrightarrow{B}_{j+1..j+l}\right)\right) = \exists_{c \in \overrightarrow{C}_{[j]}} g\left(\overrightarrow{a}, c, \overrightarrow{C}_{j+1..j+l}\right)\right)$$

We can split the quantifiers into two groups: $\overrightarrow{B}[j] = \overrightarrow{B}[j] \downarrow_{[\mathcal{U}[i+j]]^{\overline{a}}} \cup \left(\overrightarrow{B}[j] \backslash \overrightarrow{B}[j] \downarrow_{[\mathcal{U}[i+j]]^{\overline{a}}}\right)$. For the cases where c is in the second group, we know the composition of f and g will evaluate to $\bot$, regardless of the value of $\overrightarrow{B}_{j+1..j+l}$ because $\mathcal{V}$ is a valid window and $\overrightarrow{a} + (c)$ is outside of $[\mathcal{V}]^{\overline{a}}$ because $\mathcal{V}$ is contained inside $\mathcal{U}$. The equation above can therefore be simplified to:

$$\left(\overrightarrow{B} \downarrow_{[\mathcal{U}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{U}]^{\overline{a}}}\right) \Rightarrow \left(\left(\exists_{c \in \overrightarrow{B}_{[j]}\downarrow_{[\mathcal{U}]^{\overline{a}}}} g\left(\overrightarrow{a}, c, \overrightarrow{B}_{j+1..j+l}\right)\right) = \exists_{c \in \overrightarrow{C}_{[j]}\downarrow_{[\mathcal{U}]^{\overline{a}}}} g\left(\overrightarrow{a}, c, \overrightarrow{C}_{j+1..j+l}\right)\right)$$

Because (1) $\mathcal{V}$ is contained in $\mathcal{U}$, (2) $\mathcal{V}$ is a valid window for the composition of f and g, (3) the composition of f and g is equivalent to the value of g (by Theorem 5.2.3), we know the values of g in the equation above must evaluate to the same values for both $\overrightarrow{B}$ and $\overrightarrow{C}$ for the same value of c. But since the quantifiers have the exact same sets of computations, the two sides of the equation must evaluate to $\top$.

This proves the two conditions in Equation 5.8 and Equation 5.9, proving the theorem. □

Theorem 5.2.11. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \times \mathbb{B}^k \to \mathbb{B}$ be a predicate in DNF over i input sets and j context sets, dependnt on k boolean variables. Let $g : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B}$ be a predicate build from f by removing any constraints that have any of the k boolean variables. Let $\mathcal{W} : \mathbb{I}^{i+j}$ be a valid window for g. Then $\mathcal{W}$ is a valid window for f, for each combination of the boolean variables, that is, $\forall_{\overrightarrow{q} \in \mathbb{B}^k} \text{valid}\left(\mathcal{W}, \lambda_{\overrightarrow{a}:\mathbb{C}^i, \overrightarrow{B}:(\mathbb{PC})^j}.f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right)\right)$.*

Proof. To show $\mathcal{W}$ is a valid window for f, we must show that:

$$\forall_{\overrightarrow{q} \in \mathbb{B}^k} \forall_{\overrightarrow{a} \in \mathbb{C}^i} \forall_{\overrightarrow{B} \in (\mathbb{PC})^j} \forall_{a \in \overrightarrow{a}} \overrightarrow{a}\overrightarrow{a} \notin [\mathcal{W}]^{\overline{a}} \Rightarrow \left(f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right) \Leftrightarrow \bot\right)$$
$$\forall_{\overrightarrow{q} \in \mathbb{B}^k} \forall_{\overrightarrow{a} \in \mathbb{C}^i} \forall_{\overrightarrow{B} \in (\mathbb{PC})^j} \forall_{\overrightarrow{C} \in \mathbb{C}^j} \forall_{a \in \overrightarrow{a}} \left(\overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overline{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{W}]^{\overline{a}}}\right) \Rightarrow \left(f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right) \Leftrightarrow f\left(\overrightarrow{a}, \overrightarrow{C}, \overrightarrow{q}\right)\right)$$

If $\mathcal{W}$ is a valid window for g, then for any $\overrightarrow{a}$ such that $\overrightarrow{\overrightarrow{a}\overrightarrow{\notin}}\ [\mathcal{W}]^{\overrightarrow{a}}$, g evaluates to $\perp$. But for that to happen it must be the case that at least one constraint evaluates to $\perp$. This proves the first condition.

To show the second condition, we must show that g being the same for two sets B and B impllies f is the same. Let's assume this is not the case, that there is a $\overrightarrow{q}$, a $\overrightarrow{a}$, a $\overrightarrow{B}$, a $\overrightarrow{C}$ and a a, such that $\overrightarrow{B} \downarrow_{[\mathcal{W}]^{\overrightarrow{a}}} = \overrightarrow{C} \downarrow_{[\mathcal{W}]^{\overrightarrow{a}}}$ and $g\left(\overrightarrow{a}, \overrightarrow{B}\right) = g\left(\overrightarrow{a}, \overrightarrow{C}\right)$, but for which $f\left(a, \overrightarrow{B}, \overrightarrow{q}\right) \neq f\left(a, \overrightarrow{C}, \overrightarrow{q}\right)$. Since $\neg g\left(\overrightarrow{a}, \overrightarrow{B}\right) \Rightarrow \neg f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right)$ (same reasoning as for the first equation to prove), this can only happen if $g\left(\overrightarrow{a}, \overrightarrow{B}\right) \Leftrightarrow \top$ and $f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right) \Leftrightarrow \perp$ (or with $\overrightarrow{C}$ instead of $\overrightarrow{B}$, but the situation is symmetrical so proving one case proves both). But if $g\left(\overrightarrow{a}, \overrightarrow{B}\right)$ is $\top$ it means every constraint in the g must be $\top$. Since all constraints with any of the boolean variables are removed, if $f\left(\overrightarrow{a}, \overrightarrow{B}, \overrightarrow{q}\right)$ is $\perp$ it is because one of the added constraints is $\perp$. But if the added constraint is $\perp$, then there is no way $f\left(\overrightarrow{a}, \overrightarrow{C}, \overrightarrow{q}\right) \Leftrightarrow \top$, because it has the same exact constraint. This proves the second condition, proving that if $\mathcal{W}$ is a valid window for g, it is a valid window for f. □

Theorem 5.2.12. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B}$ be a predicate and $\mathcal{W} : \mathbb{I}^{i+j}$ a valid window for f. Let $I : \mathbb{I}$ be any interval. Then the window $\mathcal{V} : \mathbb{I}^{i+j}$ that is equal to $\mathcal{W}$ for all dimensions except k, and such that $\mathcal{V}[k] = \mathcal{W}[k] \cup_I I$ is also a valid window for f.*

Proof. We need to prove both Equation 5.8 and Equation 5.9 for $\mathcal{V}$. If the f evaluates to $\perp$ for any value outside $\mathcal{W}$, it will evaluate to $\perp$for any value outside $\mathcal{V}$, as $\mathcal{V}$ is a superset of $\mathcal{W}$, proving Equation 5.8. If the projections of both B and C are equal in $\mathcal{V}$, they are equal in $\mathcal{W}$, proving Equation 5.9. □

Theorem 5.2.13. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B}$ be a predicate over i input sets and j context sets. Let $g : i \times j - 1 \to \mathbb{B}$ be a predicate that uses the same twice as a context set for f: $g\left(\overrightarrow{a}, \overrightarrow{B}\right) = f\left(\overrightarrow{a}, (\overrightarrow{B}[0]) + \overrightarrow{B}\right)$ (we assume it is the first one, but because the sets can be freely reordered, this has no loss in generality). Let $\mathcal{W} : \mathbb{I}^{i+j}$ be a valid window for f. Then the window $\mathcal{W}_{0..i-1} + (\mathcal{W}[i] \cup_I \mathcal{W}[i+1]) + \mathcal{W}_{i+2..i+j-1}$ is a valid window for g.*

Proof. This is an application of Theorem 5.2.12 whereby element $\mathcal{W}[i]$ is extended with $\mathcal{W}[i+1]$ and vice-versa. □

In this section we have shown how to construct a valid window for a recognizer whose selection predicate contains quantifiers by starting with windows for non-quantified formulas and incrementally adding quantifiers.

5.2.5 Buffered Recognizer Operation

Computing windows for recognizers, as described in the previous sections, allows determining windows for recognizers, but doesn't show *how* these recognizers can operate.

We propose an implementation where every recognizer operates independently from other recognizers and maintains sets of computations in its internal state. The recognizers are connected to each other and "feed" output computations to the inputs and contexts of other recognizers, like in the example shown in Figure 5.1. Each recognizer has a buffer – a set of computations – for each input and context. These sets are subsets of the input and context sets defined in Section 5.1.3. The contents of each the recognizer's internal state sets change over time as new computations are added by probes or other recognizers and as computations in the set are, as we will see later, removed by the recognizer. In this section we will formalize this implementation and show that, for recognizers with limited valid windows (as defined in Theorem 5.2.1), the number of computations held in the buffers is finite. We will also show how the windows may introduce delay in the operation of the recognizers, providing the key principles for the discussion of delays in Section 5.2.8. At the end of this section we will have an implementable algorithm for recognizers. This algorithm can still be improved and in Section 5.2.6 further refinement of the algorithm will be discussed to deal with certain cases of unlimited windows.

To formalize the implementation of a recognizer, we start by defining a *temporal sequence* of a set of computations. A temporal sequence of a set of computations is a mapping from time to a subset of the set of computations. It represents a set of computations changing over time. We represent a sequence as: $\widehat{a}$. All sequences have the same type, $\mathbb{T} \rightarrow \mathbb{P}\mathbb{C}$.

We say a sequence $\widehat{a}$ is *smooth* if it verifies Equation 5.17. A smooth sequence is a sequence where computations "join" in at some point and later they "leave" the sequence.

$$\forall_{t\in\mathbb{T}}\forall_{u\in\mathbb{T}|u>_t t}\forall_{b\in\mathbb{C}} b \in \widehat{a}\,(t) \wedge b \in \widehat{a}\,(u) \Rightarrow \left(\forall_{v\in\mathbb{T}|v>_t t\wedge v<_t u} b \in \widehat{a}\,(v)\right) \tag{5.17}$$

Unless explicitly stated otherwise, all sequences discussed in this document are smooth.

A *spread* of a set is a sequence where the computations that *can* be at each time-indexed set are those that have completed by the indexing time or a bit further into the future. We call this extension into the future a *lookahead*. If there is no lookahead then only present and past computations may be in each set. This future lookahead is responsible for the delays discussed in Section 5.2.8.

Formally, we say a sequence is a *spread* of a set with lookahead t if:

1. All computations in the set are present at least at the time index corresponding to their endtimes (see Equation 5.18).
2. Each set in the sequence is a subset of the set (see Equation 5.19). While this condition in itself would allow a computation in the set not appear in the sequence, this is not possible due to the previous condition.obligatory
3. No computations are present in the set indexed by u if they end after $t +_t u$ (see Equation 5.20). This means lookahead in the sequence is limited to t.
4. The sequence is smooth as defined in Equation 5.17.

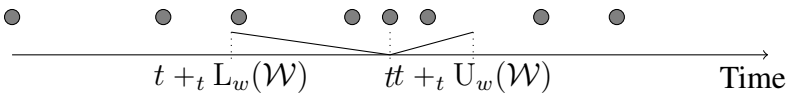


Figure 5.6: Illustration of a window-based spread of a set of computations

$$\mathrm{spread}_t\left(\widehat{a}, A\right) \Rightarrow \left(\forall_{b \in A} b \in \widehat{a}\left(\overline{b} -_t t\right)\right) \tag{5.18}$$

$$\mathrm{spread}_t\left(\widehat{a}, A\right) \Rightarrow \left(\bigcup_{u \in \mathbb{T}} \widehat{a}\left(u\right) = A\right) \tag{5.19}$$

$$\mathrm{spread}_t\left(\widehat{a}, A\right) \Rightarrow \left(\forall_{b \in A} \forall_{u \in \mathbb{T} | u <_t \overline{b} -_t} b \notin \widehat{a}\left(u\right)\right) \tag{5.20}$$

We write sequences of vectors of computations as $\overrightarrow{\widehat{a}}$ and we extend the spread operator to allow checking that each sequence in the set is a spread of a set in a vector of computations.

$$\mathrm{spread}_t\left(\overrightarrow{\widehat{a}}, \overrightarrow{A}\right) = \left(\forall_{i \in 0..\# \overrightarrow{\widehat{a}} - 1} \mathrm{spread}_t\left(\overrightarrow{\widehat{a}}[i], \overrightarrow{A}[i]\right)\right) \tag{5.21}$$

Given a vector of sets of computations, $\overrightarrow{A} : (\mathbb{PC})^i$ and a window $\mathcal{W} : \mathbb{I}^i$, both with i dimensions, we can define a *window-based spread* of the set, $\mathrm{wspread}_{\mathcal{W}}\left(\overrightarrow{A}\right)$. This is a spread built by filtering the set at any time with the given window as in Equation 5.22.

$$\forall_{\mathcal{W} \in \mathbb{I}^i} \forall_{\overrightarrow{A} \in (\mathbb{PC})^i} \mathrm{wspread}_{\mathcal{W}}\left(\overrightarrow{A}\right) = \lambda_{t:\mathbb{T}}.\overrightarrow{A} \downarrow_{[\mathcal{W}]^t} \tag{5.22}$$

Theorem 5.2.14. *Let $\mathcal{W}$ be a window with i dimensions, let $\overrightarrow{A}$ be a vector of i sets of computations. The sequence defined by* $\mathrm{wspread}_{\mathcal{W}}\left(\overrightarrow{A}\right)$ *is a spread of $\overrightarrow{A}$ with lookahead* $\mathrm{U}_w(\mathcal{W})$.

Proof. Trivial from the definition of window and spread. □

With the definition window-based spread we can now build spreads of all input and context sets of a recognizer (formally defined in Equation 5.1) as long as that recognizer has a valid window $\mathcal{W}$.

We introduce the concept of a timed selection predicate. A timed selection predicate is a parametrization of a selection predicate that at some point $t : \mathbb{T}$ only accepts sets of computations that where the latest one has terminated at that time moment.

$$\forall_{t \in \mathbb{T}} \forall_{\overrightarrow{a} \in \mathbb{C}^i} \forall_{\overrightarrow{B} \in (\mathbb{PC})^j} f^{(t)}\left(\overrightarrow{a}, \overrightarrow{B}\right) = f\left(\overrightarrow{a}, \overrightarrow{B}\right) \wedge M(\overrightarrow{a}) = t \tag{5.23}$$

Theorem 5.2.15. *Let $R_{f,e}$ be a recognizer with i inputs and j context sets. Let $\mathcal{W} : \mathbb{I}^{i+j}$ be a window for the recognizer that verifies the conditions of Theorem 5.2.1. Let $\overrightarrow{A} : (\mathbb{PC})^i$ be a*

vector of input sets of the recognizer and let $\overrightarrow{\widehat{b}} : (\mathbb{PC})^j$ *be a vector of context sets. Let* C *be the output of the recognizer as defined in Equation 5.24. Let* $\widehat{c}$ *be defined by Equation 5.25. Then Equation 5.26 is* $\top$.

$$C = R_{f,e}\left(\overrightarrow{A}, \overrightarrow{B}\right) \tag{5.24}$$

$$\forall_{t \in \mathbb{T}} \widehat{c}(t) = R_{f^{(t)},e}\left(\text{wspread}_{\mathcal{W}}\left(\overrightarrow{A}\right)(t), \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right) \tag{5.25}$$

$$\bigcup_{t:\mathbb{T}} \widehat{c}(t) = C \tag{5.26}$$

Proof. Consider any $\overrightarrow{a} \in \overrightarrow{A}$. If $f\left(\overrightarrow{a}, \overrightarrow{B}\right) = \top$, then we know, by Theorem 5.2.1, that for any $a \in \overrightarrow{a}$, $f\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(\overline{a})\right) = \top$. Therefore, due to the definition of timed selection predicate in Equation 5.23, for $t = M(\overrightarrow{a})$, $f^{(t)}\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right) = \top$. But if $\overrightarrow{a}$ is accepted at time $M(\overrightarrow{a})$, then $e(\overrightarrow{a}) \in \widehat{c}(M(\overrightarrow{a}))$. Because we assumed $f\left(\overrightarrow{a}, \overrightarrow{B}\right) = \top$, we know $e(\overrightarrow{a}) \in C$. This means all computations accepted by the recognizer in Equation 5.24 will be accepted by the recognizer defined in Equation 5.25.

Let's consider the case where $f\left(\overrightarrow{a}, \overrightarrow{B}\right) = \bot$. For any $t \neq M(\overrightarrow{a})$, we know $f^{(t)}\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right) = \bot$, by Equation 5.23. So $\overrightarrow{a}$ will never be accepted by the recognizer in Equation 5.25 at any time other than $M(\overrightarrow{a})$.

For $t = M(\overrightarrow{a})$, we know, by Theorem 5.2.1, that $f\left(\overrightarrow{a}, \overrightarrow{B}\right) = f\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right)$, since $\mathcal{W}$ is a valid window for the recognizer. With Equation 5.23 we know $f\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right) = f^{(t)}\left(\overrightarrow{a}, \text{wspread}_{\mathcal{W}}\left(\overrightarrow{B}\right)(t)\right)$. This means $\overrightarrow{a}$ will never be accepted by the timed selection predicate at any time and, therefore, $e(\overrightarrow{a}) \notin C$, proving the theorem. □

Theorem 5.2.15 is a refined version of Theorem 5.2.1. It specifies that if a set of input computations is accepted by the selection predicate, then it will be accepted at the time of the last computation in the vector. This provides a simple algorithm to implement a recognizer: every time a computation arrives, it is added to the recognizer and tested against all other input computations for acceptance by the filter. Computations are removed from the recognizer's buffers when they reach the end of the window as the window moves forward in time.

The average number of computations stored by the recognizer can be computed as the window size multiplied by the maximum rate at which computations arrive multiplied by the number of inputs. The computational complexity of adding a computation to the recognizer can be estimated to be in the order or magnitude number of the combinations of computations stored in its input buffers.

This provides the basis for the algorithms that make up the recognizer's operational logic.

Future lookahead is dealt with by introducing a delay. At time t the value of $\text{wspread}_{\mathcal{W}}\left(\overrightarrow{A}\right)$ is not not available if $\text{U}_w(\mathcal{W}) >_t 0s$. However, at time $t +_t \text{U}_w(\mathcal{W})$ it will be available. The same

Algorithm 1 Algorithm executed when a computation in an input is detected.

function INPUT_RECEIVED(i, a) ▷ Computation a detected at input i at time t.
  $\lambda_1 \leftarrow \lambda(\text{INPUT\_RECEIVED\_NOW}(i, a))$
  SET_ALARM($\mathrm{U}_w(\mathcal{W}), \lambda_1$) ▷ Process after the lookahead delay.
end function
function INPUT_RECEIVED_NOW(i, a)
  $\overrightarrow{A}[i] \leftarrow \overrightarrow{A}[i] \cup \{a\}$
  $\lambda_1 \leftarrow \left(\overrightarrow{A}[i] \leftarrow \overrightarrow{A}[i] \backslash \{a\}\right)$
  SET_ALARM($-_t \mathrm{L}_w(\mathcal{W}), \lambda_1$) ▷ Remove computation after window.
  for all $\overrightarrow{a} \in \times\left(\overrightarrow{A}\right)$ **do**
    if $f\left(\overrightarrow{a}, \overrightarrow{B}\right)$ **then**
      $e_1 = e\left(\overrightarrow{a}\right)$
      EMIT(e_1) ▷ Emit result if any combination matches.
    end if
  end for
end function

applies for the context sets. A recognizer can hold future events in its buffers if its internal clock is set to the past. This is the source of the delay discussed in more detail in Section 5.2.8.

Algorithm 2 Algorithm executed when a computation in a context is detected.

function CONTEXT_RECEIVED(i, b) ▷ Computation b detected at context i.
  $\overrightarrow{B}[i] \leftarrow \overrightarrow{B}[i] \cup \{b\}$ ▷ Add the computation to the context set.
  $\lambda_1 \leftarrow \left(\overrightarrow{B}[i] \leftarrow \overrightarrow{B}[i] \backslash \{b\}\right)$
  ALARM_AT(λ_1) ▷ Remove the computation once it leaves the window.
end function

The algorithms above describe the operation of a recognizer that keeps computations in a buffer during at least a time window. When new computations are placed in the buffer we match them against existing computations in the other buffers. We showed such operation is proven correct and, provided the window is finite, that it requires a limited number of computations implying it is computationally limited and memory limited.

But there are two issues not addressed by the mechanism described above: it requires the window to be finite for the memory and computation complexity to be finite, and being limited doesn't mean being efficient. In the remainder of this section, and in the following ones, we will discuss *purging buffers*, that is, removing useless computations from buffers that do not affect the final result. We show that purging buffers allows some important cases of recognizers with infinite windows to operate with a finite number of computations in memory and can provide for a more efficient execution.

Purging a buffer is essentially removing computations whose presence in the buffer has no

effect on the evaluation of the selection predicate. This purge is time-sensitive: at some time t a computation a may be useful but at time u (with $u >_t t$) it may no longer be useful.

Let's illustrate this with an example. Consider a selection predicate for a recognizer that matches two computations, an open with a close, such that each open is matched to a close as long as they have the same file descriptor and that there are no other open or close computations with those file descriptors in between. Such a recognizer uses two input sets, A and B, where it draws open and close computations from. It also uses the two similar sets as context sets because it needs to look to other opens and other closes to decide whether to match an open to a close as will be seen below. The selection predicate for such a recognizer could be written as in Equation 5.27:

$$f\left(a, b, A, B\right) = \texttt{fd}(a) = \texttt{fd}(b) \wedge \overline{a} <_t \overline{b} \wedge X\left(a, b, A\right) \wedge X\left(a, b, B\right) \tag{5.27}$$

$$X\left(a, b, C\right) = \neg \left(\exists_{c \in C} \texttt{fd}(a) = \texttt{fd}(c) \wedge \overline{c} >_t \overline{a} \wedge \overline{c} <_t \overline{b}\right) \tag{5.28}$$

Equation 5.27 states that an open a matches a close b if they have the same file descriptor, the open comes before the close and there are no more opens or closes for the same file descriptor in between. These last conditions prevent matching opens with anything other than the immediately following close for the same file descriptor.

If we apply the rules described in Section 5.2.3, the window for this predicate is $\mathbb{T}^4$. This can be seen intuitively as there are no restrictions in the predicate to limit the time a close can take to appear after an open. But we know intuitively that we don't need to keep around opens and closes that have been already matched, even though the predicate doesn't explicitly refer this restriction.

Suppose now that we did find an open computation d and a close computation e that verify the selection predicate, that is, $f\left(d, e, A, B\right) = \top$. If the current time is v, we know that $v >_t \overline{e}$ and $v >_t \overline{d}$ as we can only recognize computations in the present or in the past. If we could prove that there is no future computation a such that $\overline{a} >_t v$ and that $f\left(a, e, A, B\right) = \top$ we could remove e from the recognizer's input buffer (we will provide a formal proof of this later; for now let's follow intuitively). Similarly, if we could prove that there is no future computation b such that $\overline{b} >_t v$ such that $f\left(d, b, A, B\right) = \top$, we could remove d from the input buffer.

Formalizing (and generalizing) the reasoning presented above requires reasoning about future computations, to be able to reproduce the"there is no future computation such that" reasoning intuitively exposed above. This will be done in the following section.

5.2.6 Future Values

Given a selection predicate $f : \mathbb{C}^i \times (\mathbb{P}\mathbb{C})^j \to \mathbb{B}$, a vector of input sets $\overrightarrow{A} : (\mathbb{P}\mathbb{C})^i$ and a vector of context sets, $\overrightarrow{B} : (\mathbb{P}\mathbb{C})^j$, we say a computation a of set $k (k \geqslant 0 \wedge k < i)$ is useless at time t if:

$$\forall_{\overrightarrow{a} \in \overrightarrow{A}_{-k} | M(\overrightarrow{a}) \geqslant_t t} f\left(\overrightarrow{a} +_k (a), \overrightarrow{B}\right) \Leftrightarrow \bot \tag{5.29}$$

Intuitively, a computation is useless at a time if there are no combinations of input computations, with at least one ending at least at that time, that can be accepted by the selection

predicate. In the model described in Section 5.2.5, this allows removing the computation from the recognizer's buffers even if it still fits in the window.

For example, consider the following selection predicate:

$$f\left(a,b\right)=\overline{a}<_{t}\overline{b} \tag{5.30}$$

This predicate matches any a to any b as long as the second comes before the first. Consider a computation c in the set matching the second parameter of f, such that $t=\overline{c}$. At time t, this computation cannot match any new computation that may arrive in the first parameter so it is useless. This is captured by Equation 5.29 as any $\overrightarrow{a}$ that has c as second computation will evalute to $\bot$ because the first computation will have end time greater or equal to t.

Given a selection predicate, $f:\mathbb{C}^{i}\times\left(\mathbb{PC}\right)^{j}\rightarrow\mathbb{B}$, a vector of input sets, $\overrightarrow{A}:\left(\mathbb{PC}\right)^{i}$, a vector of context sets, $\overrightarrow{B}:\left(\mathbb{PC}\right)^{j}$, and a computation that belongs to a context set $k\in0..j-1$, $b\in\overrightarrow{B}[k]$, we say computation b is useless at time $t:\mathbb{T}$ if:

$$\forall_{\overrightarrow{a}\in\overrightarrow{A}_{|M(\overrightarrow{a})\geqslant_{t}t}}f\left(\overrightarrow{a},\overrightarrow{B}\right)\Leftrightarrow f\left(\overrightarrow{a},\overrightarrow{B}_{-k}+_{k}\left(\overrightarrow{B}[k]\backslash\{b\}\right)\right) \tag{5.31}$$

Just like previously for input sets, a computation b is useless at time t if its presence is irrelevant to the recognition of any sets of computations that contain at least one computation coming after t.

The definition of useless does not make any reference to time windows. That is important because it allows removing computations from buffers even when the windows are unlimited. The definitions above allow us to remove computations from input sets and from context sets, but they do not provide us with any way of knowing whether a computation is useless at a certain t without access to the full sets.

It is not possible to say whether an individual computation a is useless at some point in time in isolation. It is usually possible to say it is useless because the computations that have arrived before $\overline{a}$ are such that the selection predicate will always evaluate to $\bot$ for any $\overrightarrow{a}$ that contains a. We encode this concept in a transformation of the selection predicate that determines if it is possible for the selection predicate to evaluate to $\top$.

The useless transformation comes in two flavors, one with a *bias towards* $\top$, $U^{\top}$ and one with a *bias towards* $\bot$, $U^{\bot}$. These transformations operate on a predicate and receive two additional parameters: an index of a set to remove and a list of indexes input sets to consider to be future. So, the useless transformation biased towards $\top$ of f for set 0 with input sets $\{1,2\}$ is written as ${U^{\top}}_{0,\{1,2\}}\left[f\right]$. A useless transformation with $\overrightarrow{l}$ input sets indexes marked as future applied to a predicate with type $\mathbb{C}^{i}\times\left(\mathbb{PC}\right)^{j}\rightarrow\mathbb{B}$ is a function with type $\mathbb{C}\times\mathbb{T}\times\mathbb{T}^{\#l}\rightarrow\mathbb{B}$. The resuting function only depends on the computation to test, the current time and $\#\overrightarrow{l}$ time values that represent how far in the future each of the $\#\overrightarrow{l}$ computations will arrive. It does depends neither on i nor on j.

The useless transformation is defined by construction. In general, given a formula $f:\mathbb{C}^{i}\times\left(\mathbb{PC}\right)^{j}\rightarrow\mathbb{B}$ in DNF with no negated quantifiers, we define the transformation ${U^{\top}}_{k,\overrightarrow{l}}\left[f\right]$ as:

1. For each term that is not a quantifier:
   (a) If the term depends on anything other than the end time of input to remove or the end time of one of the future inputs, replace the term with the bias ($\top$ for $U^{\top}$ and $\bot$ for $U^{\top}$) and there's nothing else to do the term.

(b) Replace the end time of input l with $t +_t \delta_l$.

2. For each term that is a quantifier over set $\overrightarrow{B}[m]$, replace the quantifier with:

(a) For a $\exists_{c \in \overrightarrow{B}_{[m]}} g\left(a, c, \overrightarrow{B}\right)$ in a $U^{\top}$ transformation:

$$\left(\bigvee_{c:\{c:\overrightarrow{B}_{[m]} | \overline{c} <_t t\}} {U^{\top}}_{k,\overrightarrow{l}} \left[\lambda_{\overrightarrow{a}:\mathbb{C}^i, \overrightarrow{B}:(\mathbb{PC})^j} . g\left(\overrightarrow{a}, c, \overrightarrow{B}\right) \right] X \right) \wedge {U^{\top}}_{k,\overrightarrow{l}_{\cup\{m\}}} [g] X$$

(b) For a $\exists_{c \in \overrightarrow{B}_{[m]}} g\left(a, c, \overrightarrow{B}\right)$ in a $U^{\perp}$ transformation:

$$\bigvee_{c:\{c:\overrightarrow{B}_{[m]} | \overline{c} <_t t\}} {U^{\top}}_{k,\overrightarrow{l}} \left[\lambda_{\overrightarrow{a}:\mathbb{C}^i, \overrightarrow{B}:(\mathbb{PC})^j} . g\left(\overrightarrow{a}, c, \overrightarrow{B}\right) \right] X$$

(c) For a $\forall_{c \in \overrightarrow{B}_{[m]}} g\left(a, c, \overrightarrow{B}\right)$ in a $U^{\top}$ transformation:

$$\bigwedge_{c:\{c:\overrightarrow{B}_{[m]} | \overline{c} <_t t\}} {U^{\perp}}_{k,\overrightarrow{l}} \left[\lambda_{\overrightarrow{a}:\mathbb{C}^i, \overrightarrow{B}:(\mathbb{PC})^j} . g\left(\overrightarrow{a}, c, \overrightarrow{B}\right) \right]$$

(d) For a $\forall_{c \in \overrightarrow{B}_{[m]}} g\left(a, c, \overrightarrow{B}\right)$ in a $U^{\perp}$ transformation:

$$\left(\bigwedge_{c:\{c:\overrightarrow{B}_{[m]} | \overline{c} <_t t\}} {U^{\perp}}_{k,\overrightarrow{l}} \left[\lambda_{\overrightarrow{a}:\mathbb{C}^i, \overrightarrow{B}:(\mathbb{PC})^j} . g\left(\overrightarrow{a}, c, \overrightarrow{B}\right) \right] \right) \vee {U^{\perp}}_{k,\overrightarrow{l}_{\cup\{m\}}} [g]$$

Intuitively, the $U^{\top}$ and $U^{\perp}$ transformations will change the predicate to one that only depends on end times of computations that may happen in the future (the k computations). The transforamtion biased towards $\top$ will assume that terms it cannot compute are $\top$, will assume existential quantifiers may eventually quantify over something that will make the formula $\top$, and will assume universal quantifiers won't eventually quantify over anything that may be $\perp$. Conversely, transformation biased towards $\perp$ will assume the inverse.

For example, consider the example given in Equation 5.30. We can compute the useless transformation biased towards $\top$ for the first parameter, ${U^{\top}}_{0,\{1\}}[f] = t <_t t +_t \delta_b$. We will later make use of the fact that, because this equation has a solution with $\delta_b >_t 0$ it means that, if we assume a computation a such that $\overline{a} = t$, a computation b can come in the future that matches computation a.

However, if we consider for the second parameter, ${U^{\top}}_{1,\{0\}}[f] = t +_t \delta_b <_t t$, we see this only has solutions with $\delta_b <_t 0$, meaning it can only match computations in the past. This verifies what was intuitively seen about the predicate in Equation 5.30: b computations can be discarded immediately, but a computations can not.

The importance of the useless transformation is derived from Theorem 5.2.16 and Theorem 5.2.17. Theorem 5.2.16 establishes that we can determine that a computation is useless (see Equation 5.29 and Equation 5.31) from the value of the useless transformation. Theorem 5.2.17 establishes that a useless computation can be removed in a buffered operation as described in Section 5.2.5.

Theorem 5.2.16. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B}$ be a predicate over i computations ($i > 1$) and j sets of computations. Let f be such that it has exactly one quantified subexpression per context set, which may be nested inside other quantified subexpressions. f and all its quantified sub-formulas are in DNF. Let $\overrightarrow{A} : (\mathbb{PC})^i$ be a set of inputs and $\overrightarrow{B} : (\mathbb{PC})^j$ be a set of contexts. Let a be a computation in $\overrightarrow{A}[k]$, where $k \in 0..i-1$. Let $t : \mathbb{T}$ be such that $t >_t \overline{a}$. We say a is useless at time t (Equation 5.29 holds) if:*

$$\forall_{l \in 0..i-1 | l \neq k} \exists_{\delta_l \in \mathbb{T} | \delta_l >_t 0} U^{\top}{}_{i,\{l\}} \left[f\right](a, t, \{\delta_l\}) \Leftrightarrow \bot$$

Similarly, let b be a computation in $\overrightarrow{B}[m]$, where $m \in 0..j-1$. Let $\overrightarrow{i} : \mathbb{Z}^n$ be the indexes of the positions in $\overrightarrow{B}$ that are used by quantifiers enclosing $\overrightarrow{B}[m]$. Let $g : \mathbb{C}^{i+n+1} \times (\mathbb{PC})^{j-n-1} \to \mathbb{B}$ be the formula inside the quantifier over $\overrightarrow{B}[m]$, with n additional variables from the enclosing quantifiers. Let $h : \mathbb{C}^{i+n} \times (\mathbb{PC})^{j-n-1} \to \mathbb{B}$ be the conjunction of all terms of all restrictions that either contain the quantifier enclosing g or contain one of the quantifiers enclosing it. Let $u : \mathbb{T}$ such that $u >_t \overline{b}$. We say b is useless at time u (Equation 5.31 holds) if g is existentially quantified and Equation 5.32 holds of if g is universally quantified and Equation 5.33 holds.

$$\forall_{l \in 0..i-1} \exists_{\delta_l \in \mathbb{T} | \delta_l >_t 0} U^{\top}{}_{i+n+1,\, \overrightarrow{i}} \left[g \wedge h\right](b, u, \{\delta_l\}) \Leftrightarrow \bot \tag{5.32}$$

$$\forall_{l \in 0..i-1} \exists_{\delta_l \in \mathbb{T} | \delta_l >_t 0} U^{\bot}{}_{i+n+1,\, \overrightarrow{i}} \left[g \wedge h\right](b, u, \{\delta_l\}) \Leftrightarrow \top \tag{5.33}$$

Proof. First, consider transformations biased towards $\top$. If the transformation returns $\bot$, then it means all constraints have evaluated to $\bot$. This means all constraints in the predicate have at least one condition in the transformation that is $\bot$: if all conditions had been removed from a constraint, the constraint would have evaluted to $\top$ and the transformation would unconditionally evalute to $\top$.

For each condition that has evaluated to $\bot$, we can check whether it was a quantifier or not. If the condition was not a quantifier, then the predicate must also return $\bot$ because the condition is not modified. If the condition was an existential quantifier and evaluated to $\bot$, then it means that the evaluation of the quantifier condition was false for every element a such that $\overline{a} <_t t$ and that the transformation of the quantifier condition, biased towards $\top$, also returns $\bot$. If the condition was a universal quantifier, then at least one current element evaluates to $\bot$. In the case of either quantifier the evaluation of the constraint in the original predicate must always be $\bot$ because the terms still exist in the original predicate.

The reasoning for transformations towards $\bot$ is symmetrical: if the transformation returns $\top$ then we every condition must evaluate to $\top$ in the original predicate. □

Theorem 5.2.17. *Let $f : \mathbb{C}^i \times (\mathbb{PC})^j \to \mathbb{B}$ be a selection predicate with i inputs and j context sets. Let $e : \mathbb{C}^i \to \mathbb{C}$ be an emission function with i inputs. Let $\overrightarrow{A} : (\mathbb{PC})^i$ be an input, $\overrightarrow{B} : (\mathbb{PC})^j$ be a context and $C : \mathbb{C}$ be an output. Let $w : \mathbb{T}$ be a lookahead value and $\overrightarrow{\widehat{a}} : \mathbb{T} \to (\mathbb{PC})^i$ be a*

sequence such that $\text{spread}_w\left(\overrightarrow{\widehat{a}}, \overrightarrow{A}\right)$, $\overrightarrow{\widehat{b}} : \mathbb{T} \to (\mathbb{PC})^j$ *be a sequence such that* $\text{spread}_w\left(\overrightarrow{\widehat{b}}, \overrightarrow{B}\right)$ *and* $\widehat{c} : \mathbb{T} \to \mathbb{PC}$ *be a sequence such that* $\text{spread}_w(\widehat{c}, C)$. *Let* $\overrightarrow{\widehat{a}}$, $\overrightarrow{\widehat{b}}$ *and* $\widehat{c}$ *be such that* $\forall_{t\in\mathbb{T}}\widehat{c}(t) = R_{f^{(t)},e}\left(\overrightarrow{\widehat{a}}(t), \overrightarrow{\widehat{b}}(t)\right)$ *and* $C = R_{f,e}\left(\overrightarrow{A}, \overrightarrow{B}\right)$.

Let $k : \mathbb{Z}$ *be a value in* $0..i-1$. *Let* $a : \mathbb{C}$ *be a computation in* $\overrightarrow{A}[k]$. *Let* $u : \mathbb{T}$ *be a point in time such that* a *is useless at* u *as defined in Equation 5.29. Let* $\widehat{d} : \mathbb{T} \to \mathbb{PC}$ *be a sequence defined as:*

$$\forall_{t\in\mathbb{T}}\left(t <_t u \Rightarrow \widehat{d}(t) = R_{f^{(t)},e}\left(\overrightarrow{\widehat{a}}(t), \overrightarrow{\widehat{b}}(t)\right)\right) \wedge \left(t \geqslant_t u \Rightarrow \widehat{d}(t) = R_{f^{(t)},e}\left(\overrightarrow{\widehat{a}}(t)\setminus\{a\}, \overrightarrow{\widehat{b}}(t)\right)\right)$$

Let $l : \mathbb{Z}$ *be a value in* $0..j-1$. *Let* $b : \mathbb{C}$ *be a computation in* $B[l]$. *Let* $v : \mathbb{T}$ *be a point in time such that* b *is useless at* v *as defined in Equation 5.31. Let* $\widehat{e} : \mathbb{T} \to \mathbb{PC}$ *be a sequence defined as:*

$$\forall_{t\in\mathbb{T}}\left(t <_t v \Rightarrow \widehat{e}(t) = R_{f^{(t)},e}\left(\overrightarrow{\widehat{a}}(t), \overrightarrow{\widehat{b}}(t)\right)\right) \wedge \left(t \geqslant_t v \Rightarrow \widehat{e}(t) = R_{f^{(t)},e}\left(\overrightarrow{\widehat{a}}(t), \overrightarrow{\widehat{b}}(t)\setminus\{b\}\right)\right)$$

Then, $\forall_{t\in\mathbb{T}}\widehat{d}(t) = \widehat{c}(t)$ *and* $\forall_{t\in\mathbb{T}}\widehat{e}(t) = \widehat{c}(t)$.

Proof. Trivial from Equation 5.29 and Equation 5.31. If the presence of the computation does not affect the selection predicate, the outputs are the same. □

With the definitions above, we can now extend the way a recognizer operates beyond what was defined in Algorithm 2 and in Algorithm 1. We can periodically scan all computations in the buffers to determine whether they are useful or not. This can occur in parallel to remove computations from the buffers even if they fit inside the time window.

5.2.7 Type System

The semantics described in this chapter consider all computations to have a single type, $\mathbb{C}$. This is a simplification of CALL because in CALL computations are typed and each type defines multiple attributes which are, themselves, also typed. And, more than just defining types of computations, CALL supports type graphs with multiple inheritance.

The lack of types in the semantics does not affect its conclusions. It only means the model used for the semantics is more abstract and less restrictive that CALL. That means all the results still apply, because if they apply in the general case, they apply in the more restrictive case.

CALL has a type verification system, its interpreter / compiler, and that verification system will ensure only type-valid statements can be made. The semantics described in this chapter only apply to programs that are well-formed.

5.2.8 Time and delays

It is possible to have a recognizer with a window whose upper bound is larger than zero. This means that, at time t, we need computations with end time $u >_t t$. CALL has no prediction system to guess whether such computations will occur and what it does is delay the clock of

each recognizer by that amount. That means all computations emitted by a recognizer with window $\mathcal{W}$ will be emitted with a delay that is equal to $\mathrm{U}_w(\mathcal{W})$ (they will still report the correct end time).

But as recognizers are used in a graph, a delay on the output of one recognizer will mean all recognizers that depend on it will get its input delayed. Since those recognizers may need to delay the output themselves, their internal clock will be delayed even further.

So, in the end, in the recognizer DAG, each recognizer will have its internal clock delayed by the sum of (1) the maximum of its recognizer window (or zero if the window is negative) and (2) the maximum delay of any of its inputs or contexts.

If we put aside limitations in the algorithm to compute windows that may lead to larger windows that strictly necessary, the delays in the recognizers are fundamental and should be reasonable for the business. For example, if we state that a ping with no response after 5 seconds is a missed ping we *must* wait for 5 seconds after the ping to report the missed ping that happened 5 seconds earlier. This is not an arbitrary limitation introduced by CALL or by the formalization of its semantics, it is implied by the problem itself.

Chapter 6

CALL Runtime

This chapter discusses the CALL runtime, also known as CALL-RE. The runtime is responsible for interpreting and analyzing CALL specifications and execute all elements in the diagnosis system's architecture with the exception of the probes. Probes are system-specific and so their lifecycle is controlled externally from CALL-RE.

CALL's runtime architecture was designed to fulfill several of the claims described in Chapter 1:

- **Improving quality by ensuring probe adequacy**. CALL specifications are sufficiently expressive to allow CALL-RE to understand which probes' outputs are used and which are not. This allows CALL-RE to flag unnecessary probes that can be removed from deployment.
- **Being general by supporting several styles of systems**. CALL-RE's architecture is independent of the monitored system's style. It can be used with single-process systems, event-based systems, repository-based systems and systems with different (and mixed) styles.
- **Being general by being independent of probe implementation technology**. CALL-RE was designed to allow probes – and, consequently, the system being monitored – to be written in any language and hardware architecture. CALL-RE was designed to even be usable in mixed systems with different probes being implemented in different technologies.
- **Being scalable with respect to system size**. CALL-RE was designed to be independent of the number of components of the system being monitored.
- **Being scalable with respect to data volume**. CALL-RE was designed to work with large data volumes, allowing its own processing to be spread among multiple servers to distribute both computation and network cost.
- **Being practical**. CALL-RE's configuration is straightforward. For small systems a few lines of CALL will do. For larger systems, CALL-RE configuration is similar, conceptually, to many other deployment systems.

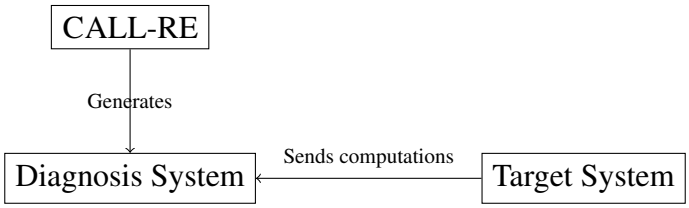


Figure 6.1: Different systems involved when executing the proposed framework.

6.1 CALL Architecture and Deployment Architecture

CALL-RE is a *runtime*: an environment that executes programs. When discussing architecture it is important to distinguish between three concepts:

- the architecture of CALL-RE itself;
- the architecture of the system that is executed when CALL-RE is given a specification written in CALL – the diagnosis system;
- the architecture of the target system, the system being monitored

These three systems are illustrated in Figure 6.1.

CALL-RE's architecture is important to the discussion of how CALL-RE itself is implemented. But, for the purpose of this thesis, we care about the structure of the interpreted system. How CALL-RE is actually structured is mostly irrelevant since it does not affect the thesis' claims.

6.1.1 CALL Template Architecture

As stated above, CALL-RE is an environment that executes programs. But, unlike operating systems or the Java Virtual Machine, CALL-RE has two significant differences: it is designed to run in multiple instances and it is significantly more restrictive with respect to the structure of the programs it executes.

For first difference between CALL-RE and operating systems means essentially that CALL-RE is networked at a fundamental level: each CALL-RE instance is designed to be aware that others exist and coordinate with others the execution of a CALL specification. In this regard we can say CALL-RE is designed to work in clusters, or in a grid. CALL-RE assumes standard TCP/IP connection between nodes.

A consequence of this design decision, made to support scalability with respect to both system size and data volume, is that *nodes*, the physical execution environments of CALL-RE, appear directly in the CALL specification that will be executed. This shows one way in which CALL is different from traditional programming languages where code is designed to be executed in a single operating system and the description of inter-node communication (networking) is built in as part of the actual program, not of the operating system.

A second consequence of this design decision is that specifying deployment of a system using CALL becomes very easy as will be shown later. This makes CALL specifications practical as they provide few degrees of freedom with most of the work done automatically by CALL-RE.

The second difference between CALL-RE and operating systems, being more restrictive with respect to the structure of programs it executes, is what allows us to talk about a template architecture for CALL programs: while the specifics will differ between systems, all CALL programs have an architecture that shares some common principles.

In all systems executed by CALL-RE, there are four types of components:

- **Recognizer**. Each component of type "recognizer" corresponds to a recognizer that is specified in CALL.
- **Oracle**. Each component of type "oracle" corresponds to an oracle that is specified in CALL.
- **Fault Localizer**. A single component of this type exists.
- **Event Bus**. At least one event bus must exist, but multiple event buses can be used to spread the load. Each event bus has one or more topics. Components can subscribe to these topics to be notified when computations are published. Components can also publish computations to these topics.

Probes are, as described in the beginning of this section, considered to be part of the target system and, while they connect to the diagnosis system, they are not part of it. That's why we don't consider probes to be components in the system. They are, however, declared in CALL (see Section 6.2) so that CALL-RE knows what probes to expect.

Each component in the system is deployed in a node. All computations are sent and received using event buses: each stream (see Section 4.2.4 and Section 6.2) corresponds to a topic in an event bus. Components and probes that publish computations to that stream will publish the computations to the event bus topic and components and oracles that read streams subscribe to the corresponding topic.

The fault localizer, a single component, has to be deployed in a node with an event bus. All computations that have been classified as correct or incorrect, are sent to a single predefined topic in that event bus.

This event-based template architecture allows for significant scalability. If there is a large number of computations, streams can be assigned to different event buses, splitting both the computation and network load. This will be discussed in more detail in Section 6.4. Also, because the computation structures are defined in a platform and technology neutral way, the system executed by CALL-RE is independent of the implementation technology of the system being monitored.

Also, as the previous discussion has shown, CALL-RE's architecture is independent of the monitored system's style. It can be used with single-process systems, event-based systems, repository-based systems and systems with different (and mixed) styles. All these are abstracted away once probes reports computations to the diagnosis system.

6.1.2 Describing the Diagnosis System's Architecture

The description of the architecture of a system executed by CALL-RE is part of the system's CALL specification. A CALL specification has no structure other than the one imposed by the language and that was described in Chapter 4 and Appendix B. However, because it is a common engineering practice to separate specifications into multiple files to deal with different concerns,

we follow the same convention in this thesis with respect to CALL. The conventions used in this thesis split the CALL specification into multiple files over multiple directories:

- `binding.call`. Binding description as described in Section 6.3.
- `bootstrap.call`. Bootstrap file, includes all the others. Its contents are presented below.
- `deployment.call`. Deployment description as described in Section 6.4.
- `oracle_types/`. Directory with all oracle types.
- `oracles.call`. Instantiates all oracles. Described in Section 6.2.
- `probes.call`. Declares all probes. Described in Section 6.2.
- `recognizer_types/`. Directory with all recognizer types.
- `recognizers.call`. Instantiates all recognizers. Described in Section 6.2.
- `streams.call`. Defines all streams 6.2.
- `types/`. Directory with data types.

Files are included by using the inclusion mechanism in CALL. For example, a typical `bootstrap.call` file will look like:

```
1  // Recursively include all types. Note the string is a regular expression.
2  include all "types/.*"';
3  include all "recognizer_types/.*"';
4  include all "oracle_types/.*"';
5
6  // Include all recognizers.
7  include "recognizers.call";
8
9  // Include all oracles.
10 include "oracles.call";
11
12 // Define streams.
13 include "streams.call";
14
15 // Define streams.
16 include "streams.call";
17
18 // Binding description.
19 include "bind.call";
20
21 // Deployment description.
22 include "deployment.call";
```

Listing 6.1: Example of a typical `boostrap.call` file.

6.2 Declaring Components

In CALL all components in the architecture have to be explicitly declared. Recognizers are, by convention, declared in their own file named `recognisers.call`. This file contains all recognizers. Also, by convention, recognizer types' names have a `_r` suffix and recognizer instances' have a `_ri` suffix.

```
1 recognizer r0_ri : r0_r;
```

```
2 recognizer r1_ri : r1_r;
```

Listing 6.2: Declaring recognizers.

Oracles are, also by convention, declared in their own file named `oracles.call`. Like recognizers, oracle type names have the `_o` suffix and oracles have the name `_oi` suffix.

```
1 oracle o0_oi : o0_o;
```

Listing 6.3: Declaring oracles.

Streams are, like recognizers and oracles, declared in their own file with a set of conventions. The file is named `streams.call`. Streams don't have a type of their own. Streams are declared with the computation type that is a superclass of all computations in that stream.

```
1 stream s0_s : c0_t;
2 stream s1_s : c1_t;
3 stream s2_s : c2_t;
```

Listing 6.4: Declaring streams.

Probes are not created or started by CALL-RE, but they have to be declared to allow analysis. CALL-RE will not accept computations from probes that were not declared in the specification. Probes that are declared *can* send computations, but they are not obliged to do so. A probe that does not send events is, from CALL-RE's perspective, indistinguishable from a probe that is not running. Probes are declared with the computation types they generate.

```
1 probe p0_p : c0_t;
```

Listing 6.5: Declaring probes.

6.3 Binding Description

CALL-RE moves computations between oracles, recognizers and the fault localizer. This binding of components and connectors defines which streams probes write to, which streams recognizers read and write from and which streams oracles read from, as was described formally in Chapter 5. In CALL-RE it is defined by a *binding description*. Binding descriptions are written in CALL. For example, for the components defined above, the following binding description can be used:

```
1 bind p0_p out s0_s;
2
3 bind r0_ri in (s0_s) out s1_s;
4 bind r1_ri in (s1_s) out s2_s;
5
6 bind o0_oi in s2_s;
```

Listing 6.6: Example of binding description.

A binding description is built as a set of `bind` statements. Bind statement syntax is slightly different between probes, recognizers and oracles. This is because probes only have output, oracles input and recognizers can have multiple inputs, but only one output. The syntax for the

three types of components is shown in the example above. As was stated previously, CALL does not instantiate probes, but requires them to be declared so it can perform analysis on the system.

The binding description in a CALL specifications is sufficiently expressive to allow the CALL compiler to understand which probes' outputs are used and which are not. This allows unnecessary probes that can be removed from deployment to be flagged.

6.4 Deployment Description

The component declaration presented in Section 6.2 and the binding description presented in Section 6.3 describe the components that form the diagnosis system's architecture and how they are connected: which streams probes place computations on, which streams recognizers read from, which streams recognizers output computations to and which streams oracles read from. Nothing in this description hints at how these are physically distributed in an actual executing environment.

The separation between physical deployment and component and connector description has been well established in the software architecture literature [12, 22]. In CALL-RE case, it provides several benefits:

- **Connections do not depend on physical location**. The way probes, recognizers and oracles are connected defines how the computation abstraction and classification graph is built. This connection is independent of hardware requirements or physical deployment. If recognizer `r0_ri` identifies high-level computations of type `c1_t` from low level computations of type `c0_t`, `r0_ri` will do so regardless of *where* it is actually running, where the "producers" of computations of type `c0_t` are and where the "consumers" of computations of type `c1_t` are. Connector specifications are concerned with *correctness of specification*.
- **Physical location does not depend on connectors**. If connectors do not depend on physical location, physical deployment also doesn't depend on the connectors, assuming we can freely communicate between different deployment nodes. So if recognizer `r0_ri` outputs computations of type `c1_t`, which are used as inputs for recognizer `r1_ri`, then these computations must flow from `r0_ri` to `r1_ri`, regardless of where they are deployed. What prompts the software engineer to allocate a recognizer to a specific node is based on resource requirements and availability. Physical deployment is concerned with *resource allocation*.
- **Binding can be reused between systems**. Binding descriptions can be split into multiple files and reused. It is possible to define in CALL an HTTP specification with types, recognizers and connectors and reuse this specification in any system that uses HTTP. But deployment descriptions are usually system specific because they involve physical nodes.

To illustrate the use of deployment descriptions to generate different architectures for the same binding description, consider the system we've been using in this section with recognizers, `r0_ri` and `r1_ri`. `r0_ri` reads computations from stream `s0_s` and emits higher-level computations to stream `s1_s`. Recognizer `r1_ri` reads computations from stream `s1_s` and emits higher-level computations to `s2_s`. Oracle `o0_oi` evaluates computations from `s2_s`. If the

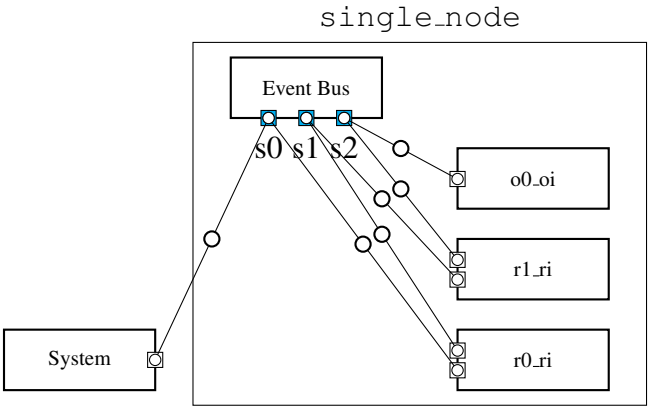


Figure 6.2: Architecture of example deployment for a low load system.

system is known upfront to have a low load (see Section 9.4.5 for a discussion on future work to change the deployment dynamically), both recognizers, the oracle and the event bus can be run in the same node:

```
1  node single_node {
2      ip "10.0.0.1";
3      // Regular expressions can be used so, in this case
4      // recognizer ".*" would also work.
5      recognizer "r0_ri";
6      recognizer "r1_ri";
7      oracle "o0_oi";
8      event_bus {
9          port 4546;
10         // Regular expressions can be used so, in this case
11         // stream ".*" would also work.
12         stream "s0_s";
13         stream "s1_s";
14         stream "s2_s";
15     }
16 }
```

Listing 6.7: Example of deployment for a low load system.

With this deployment structure, both recognizers, the oracle and the event bus are run in the same node, as seen in Figure 6.2. All three streams exist as topics in the event bus. Both recognizers use the single node's CPU and the only external node network use is due to the computations sent by the probe in the system to the event bus.

But if it is expected upfront that will be a large flow of incoming computations on stream `s0_s` and recognizer `r0_ri` will need to perform a significant amount of work, `s0_s` and `r0_ri` could be placed in their own node and a deployment with two event buses on different nodes could be used:

```
1  node s0_handling_node {
2      ip "10.0.0.1";
3      recognizer "r0_ri";
4      event_bus {
5          port 4546;
6          stream "s0_s";
7      }
8  }
```

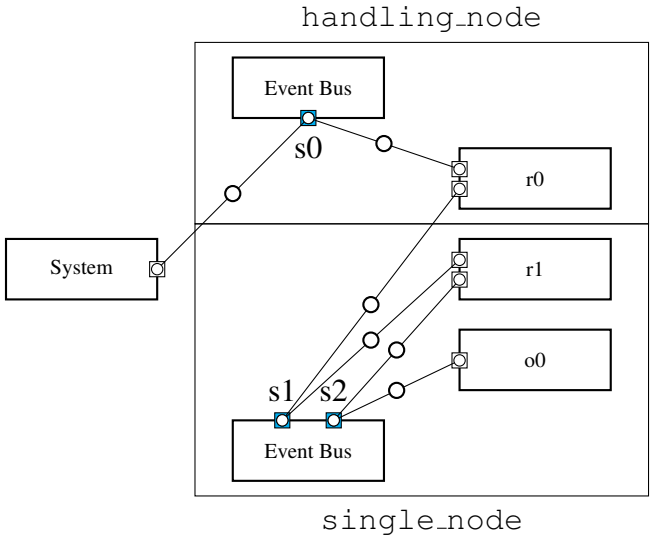


Figure 6.3: Architecture of example deployment for a system with two event buses.

```
9
10 node single_node {
11     ip "10.0.0.2";
12     recognizer "r1_ri";
13     oracle "o0_oi";
14     event_bus {
15         port 4546;
16         stream "s1_s";
17         stream "s2_s";
18     }
19 }
```

Listing 6.8: Example of deployment for a high load system.

In this case, whose architecture is depicted in Figure 6.3, the probe would send computations to stream `s0_s` in the event bus of node `s0_handling_node`. These computations would be made available to recognizer `r0_ri` running on that same node. When `r0_ri` recognizes a computation, it sends it to the topic corresponding to stream `s1_s` in the event bus of node `single_node`. This way, assuming only a few output computations are recognized by `r0_ri`, the network and computation load related to `s0_s` is placed on `s0_handling_node` exclusively.

Streams are the only level of granularity available to control routing of computations in CALL-RE. If the load on stream `s0_s` is known upfront to exceed the computation capabilities of a single physical node, then the actual system's CALL specification would have to be altered to split `s0_s` into different streams and probes would have to be configured to load balance between the streams. CALL provides no mechanism to do this type of split, usually known as *sharding*, automatically (see Section 9.4.4). However, the case studies show that CALL-RE can handle large volumes of data in a single node so sharding is only expected to be necessary in the most extreme scenarios.

A deployment description, as was shown above, is a specification written in CALL that de-

scribes the physical organization of the runtime. Deployment descriptions contain one or more CALL `node` structures. A `node` describes a physical environment where elements are deployed. Nodes can contain the following statements:

- `ip` followed by a string containing an IP address. This is used to identify how the node should be reached by oracles and recognizers that need to connect to an event bus on the node. Only nodes containing an event bus need to have an IP address. IP addresses must be unique among all nodes.
- `recognizer` followed by a string containing a regular expression that matches recognizer names. Multiple `recognizer` statements are allowed. They identify which recognizers run in the node. The same recognizer can be matched by multiple statements in the same node, but if a recognizer is matched to more than one node the specification is invalid.
- `oracle` followed by a string containing a regular expression. Just like `recognizer` statements, but for oracles.
- `fault_localizer`. A single `fault_localizer` statement must exist in only one node. This identifies the node where the fault localizer runs.
- `event_bus`. An event bus structure inside a node informs the CALL-RE that this node should run an event bus. Nodes that have event buses may also host any other components. A node with an event bus must contain an IP address. The event bus structure must contain a single `port` statement with the number of the TCP/IP port where the event bus is running and can contain multiple `stream` statements, each with a regular expression that matches the names of the streams served by the event bus. Like recognizers, the same stream can be matched multiple times in the same event bus, but if the same stream is matched to more than one event bus, the CALL specification is invalid and rejected by CALL-RE.

6.4.1 Mapping and Starting Nodes at Runtime

At runtime, CALL-RE needs to be bootstrapped individually in each node. A system administrator can easily automate this process using a script, but there far too many ways to do this and they depend on the system's configuration so CALL-RE provides no way of automatically bootstrapping each individual node.

When boostrapping CALL-RE in a node, CALL-RE must be told which node it is running on. CALL-RE has a predefined startup sequence that does not depend on the CALL specification being executed:

1. The CALL specification provided is parsed. If parsing fails then CALL-RE prints an error message and shuts down.
2. If the node specified at bootstrap time is not defined in the CALL specification, CALL-RE prints an error and shuts down.
3. If the node specified has an event bus and, therefore, an IP address, CALL-RE will verify that the current hardware it is running on has that IP address assigned. If it doesn't it prints an error message and shuts down.

4. CALL-RE is placed in `waiting` state. In this state, any computation arriving at the event bus, which has not been started yet, is discarded.
5. If the current node specifies an event bus, the event bus is started and all topics created. All computations arriving at the event bus are discarded and will continue to be discarded until CALL-RE changes state.
6. CALL-RE keeps trying to connect to all events buses required by the components registered to run in this node. It stays in this state until all event buses are reachable.
7. CALL-RE is placed in `running` state. In this state, all computations arriving at the event bus, if any, are sent to subscribers.

An interactive runtime session could look like:

```
unix> call.sh --spec=bootstrap.call --node=s0_handling_node
CALL Runtime Environment v1.0.
Initializing:
  Parsing "bootstrap.call"... done
  Starting node "s0_handling_node"... done
  node "s0_handling_node" is waiting.
Setting up wiring:
  Starting event bus at port 4546... done
  Registering stream "s0" with event bus... done
Connecting to stream "s1" in event bus 1.2.3.4:5678... failed
Connecting to stream "s1" in event bus 1.2.3.4:5678... done
Starting recognizer "r0"... done
Starting:
  node "s0_handling_node" is running.
```

Chapter 7

Related Work

7.1 Runtime fault localization in specific domains

Autonomic monitoring and diagnosis has not been done systematically and extensively [35, 67] except for typically specific domains such as image recognition [66] or intrusion detection [4].

With the proliferation of microservice applications, significant research has been done in identifying failures and localizing faults in microservice-based applications. [80] proposes dynamic monitoring of faults in micro services applications based on log processing and monitoring instead of relying on architectural models. This allows working with dynamically assembled systems, typical of micro service architectures, but provides lesser guarantees of correctness and failure localization. [65] uses call graphs to determine the faulty components in a microservice application. Our approach, however, supports non-binary failure models and systemic quality attributes which are not easily mapped to [65].

7.2 Algorithmic techniques for fault localization

Several algorithmic techniques exist that locate faults by observing failures. These techniques differ in their applicability phase (design time or run time) and their abstraction level (low level or architectural).

Design time techniques for low-level fault localization

Automatic fault localization during development time is focused on finding where bugs in software are located by observing system tests. The main purpose of these localization algorithms is to aid developers during the debugging process [59]. By observing test case runs and instrumenting code and – because test cases are classified in success or failures –, the localization algorithms will try to find out where the bugs are located within the source code.

There are two main groups of approaches for design time fault localization: model-based approaches and statistical approaches. Model based approaches [27, 54], use a model of the expected behavior of the system and compare the model with the observations. They then produce a report with the diagnosis which is used by the developers.

Statistical approaches use abstracted program traces to produce a list of fault candidates [43, 49, 50]. With less input that model-based approaches, statistical approaches produce less accurate diagnosis but have faster execution time.

A new statistical based technique – called SMFL [1, 2, 3], *spectrum-based multiple fault localization* – has been proposed. This technique is much more accurate than traditional techniques but still retains fast execution and low computational overhead.

SFL techniques have mixed results at design time. In industry the ratio between number of code blocks and tests varies significantly leading to the effectiveness of SFL varying significantly [44, 68]. This is different from our approach where the ratio between traces and architectural elements is quite large. Also, at design time, the target system is expected to contain many faults, with significant correlation, which has been found to reduce the effectiveness of SFL [29].

Some techniques are usable at design time to improve accuracy of SFL like call frequency analysis. It is common for software code to be called multiple times in a single trace and that information can help diagnosis accuracy [75]. While usable at run time, in architectural models it is less common for architectural elements to be called multiple times in traces so these techniques are less useful.

Although generally successful, these techniques are used at design time and not at run time. The use of these techniques with code blocks suffers from the problem that there is often a small proportion of tests to code reducing their effectiveness. But applying these techniques at run time with the architectural models generates high ratios of spectra per architectural element, increasing the effectiveness of SFL algorithms.

Run-time techniques that perform low-level localization or repair

A common approach in industry is to identify a class of faults at design time and prepare the system for those faults. This is mostly an *ad-hoc* way of performing diagnosis and is often coupled with local recovery. Many systems that include heartbeats or pings in their design are in this category. Google File System [34] and Hadoop [26] are two examples of such systems. But even lower level systems such as TCP/IP, with its detection of reception failure and retransmission, fall in this category.

These specific diagnosis mechanisms are usually very specialized, fast and efficient and many include *in loco* repair mechanisms. For example, if a client fails to read from a data node in a Hadoop cluster, it will automatically fetch the location of another node and will retry from the second data node.

Techniques based on this general approach to diagnosis – hand-crafted – have a few common limitations: (1) they are not portable to other systems, meaning there is very little reuse; (2) they are extremely intrusive to the system hindering maintainability; (3) they assume single-fault scenarios and the systems may behave unpredictably in case of multiple-fault scenarios; (4) they do not allow iterative implementation because the fault detection logic is deeply buried within the program code making adding support for new faults a significant engineering effort. These techniques difer from this work in two fundamental ways: they are specialized, often merging monitoring and analysis, and they include repair mechanisms.

Another set of techniques that fit into this category are dynamic code-based adaptation techniques. These try to correct software bugs automatically using program mutations like the genetic

programming [76]. Although they have proven to have some limited success, they are not predictable or reliable and are often dependent on test suites to validate correctness of the generated or mutated software.

Run-time techniques that perform high-level localization or repair

Repair-based techniques attempt to recover a system when a fault is detected, triggering a repair strategy. It is up to the repair strategy to perform any required diagnosis. For example, in the Rainbow system [21, 33], repair is triggered when any of a set of conditions is violated. Rainbow allows several strategies to be defined, but it is up to the strategies to figure out what the problem is and what has to be done to correct it. Relying on strategies to perform repair is common in self-adaptive systems. The three-layer architecture model proposed in [47], relies on high level planning to perform diagnosis after the repair has been started.

These repair-based techniques share a common set of limitations. First, they rely on the strategy designer to define the right process for diagnosis, which is in itself a complex and error-prone task. Secondly, they make reuse of diagnosis among strategies difficult. Thirdly, they force a strategy to be selected *before* knowing what is wrong. Repair strategies are no longer focused on "repair" but a mix of "diagnosis and repair".

The aforementioned systems are general self-adaptive systems which perform repair as part of the adaptation tasks. Two additional types of repair-based techniques exist: proactive techniques and dynamic code-based adaptation techniques. Proactive techniques leverage the *a priori* knowledge that systems age through time and mitigate this effect by standard techniques such as rebooting – software rejuvenation [46, 74] is one such technique. These techniques, however, do not handle all faults and have limited applicability.

Still within this approach is recovery-oriented computing [15]. This technique is based on system-wide undo support, allowed by the underlying characteristics of the JEE platform, high isolation of components – placing them in virtual machines which can be operated independently – replicating components and providing components with self-testing. Although this approach allows diagnosis and repair at a high level of abstraction, it requires specialized technology and imposes a specific architecture on the systems. To a certain extent, this approach is similar to the microservices approaches discussed earlier [65, 80].

Also addressing related issues is [38] which addresses the issues that arise from failures that are not predicted in self-adaptive systems generating instability in the adaptation loop. This work addresses, in a limited way, the problem of adapting the MAPE loop. This is also complementary to this work.

7.3 Behavior modeling and verification

There is a wide variety of behavior modeling formalisms. Formal models such as CSP [14], Z [40] or timed automata [6] have been extensively used and proven successful for many applications. These modeling formalisms are very general and applicable to a wide range of systems. In fact, they are applicable even outside the computer science domain. This generality limits their ability to match the concepts used in software architecture and provide limited reasoning in this

scenario. CALL, on the other hand, is significantly less expressive, but more easily applicable to architectural models.

Wright [5] is a formal language based on CSP that allows modeling of software architectures. Although specifically addressing software architecture, Wright is also limited in its expressiveness: it does not contain any timing information and is limited to specifying protocols and communication structure. It does provide correctness guarantees on the architectural models, whereas CALL does not.

General architecture description languages such as Acme [32, 62] or AADL [31] provide precise structures for modeling software architecture but contain limited – if any – behavior modeling capabilities.

Other modeling formalisms, of which UML [37] is the best-known, focus on informal behavior modeling, even if backed by a formal syntax. This informal modeling, although expressive, easy to use and matchable to software architecture concepts, is not capable of providing a complete, abstract, behavior model of a system. The message sequence charts used in Znn described in Section 8.1.1 are defined formally in UML.

A different class of formalism for behavior modeling is the one of *simulators*. General simulators such as Simulink [53] and probabilistic simulators such as the Prism Model Checker [48] can be used to run simulations of software architectures. More specific tools may be built using general-purpose techniques such as discrete-time event simulation [69]. However, simulators are aimed at generating behavior, not matching an observed behavior with a model and are not appropriate for diagnosis.

A last, rather obvious, category of related work in behavior modeling are programming languages such as C++ [41]. Some drawbacks of using programming languages for behavior modeling are quite obvious, such as the lack of match with software architecture concepts. However, programming languages are the most commonly used form of behavior modeling and are an important guide on how software engineers think about software systems.

All the behavior modeling tools and formalisms listed missed at least one of these fundamental requirements: (1) they must operate at the software architecture level, (2) they must distinguish good and bad behavior and (3) they must be able to match observations on a running system.

Some approaches do not model architecture, but rather model behavior and system properties. While not directly applicable, they are worth mentioning. [55] proposes techniques for runtime verification using logical formalisms. [10] addresses the problem of efficient monitoring of systems with parametetric specifications, such as the ones we use, and proposes some efficient matching algorithms based on automaton. Significant research has been done in this area of monitoring of systems based on parametric specifications. MARQ is a tool proposed in [63] that performs run-time monitoring using Quantified Event Automata. [11, 81] propose a more expressive language that allows quantification.

7.4 Complex event processing

Complex event processing (or information flow processing) is an active area of research which, broadly speaking, focuses on analyzing flows of events. [52] contains an extensive survey in

the area dividing systems in three main categories: (1) active databases, extensions to traditional relational databases which can fire actions on certain events such as data insertion or removal, (2) data stream management systems, systems which read flows (or streams) of events and try to identify patterns and (3) complex event processing systems which try to recognize broad patterns of events. There is a well-known leading open source project in this last category: Esper [30].

This thesis does not focus directly on complex event processing but builds significantly on top of it. As observed computations in the base system may not match the abstractions available in the correctness criteria, it is necessary to map the the observed computations into other computations which will be understood by the oracle. To a certain extent, CALL and CALL-RE can be seen, in part, as a complex event processing system with several unique characteristics: (1) it keeps track of the location of computations, (2) it automatically handles time windows and merging, (3) it allows for classification of events and (4) it use the classification for fault localization.

Chapter 8

Validation

Referring back to the thesis claims (Chapter 1), validation done as part this thesis has to demonstrate that all the requirements in Table 8.1 are met.

Requirements are demonstrated in three different ways:

- Arguing. Some requirements are specified informally and qualitatively. In these cases, a formal proof is not possible.
- Case Studies. Some requirements are validated through case studies. Two case studies were made whose details will be described below.
- Proving. Some requirements are validated through formal proof.

8.1 Case Studies

To validate that the proposed approach is feasible, to validate some of the requirements (more details on which below) and to gain insight into further research on specifying behavior and correctness, two case studies were performed. These case studies comprised the design of autonomic diagnosis in the Znn web system and the Samsung event-based system, both described below.

8.1.1 Znn

The Znn web system is a demo web system built by the ABLE group at Carnegie Mellon University. It has the same architecture of our motivation system which was represented in Figure 2.1. Znn is equivalent to a wide range of systems as described in Chapter 2. It is based on a typical LAMP stack (Linux, Apache, MySQL, PHP) mimicking a news site with multimedia new articles.

In this system, multiple clients access one of two dispatchers (also termed "load balancers"), which forward requests to a random web server in a farm. If the request is not for an image, the web server will access the database to fetch the required information and generate the news page with HTML text and references to images. Web clients will then access the system to fetch the

Claim	Requirement	Validation
Main Claim		
	Allows describing system behavior using a declarative approach.	Case Studies
	Allows describing what is correct and incorrect behavior also using a declarative approach.	Case Studies
	Provides mapping from failures to their sources.	Case Studies
Cost		
	Allows easy code reuse between systems.	Case Studies
	Has a low runtime probing cost.	Case Studies
Quality		
	Provides support for systemic system properties (*i.e.*, quality attributes).	Case Studies
	Guarantees system coverage.	Proving
	Systematically finds and orders failure cases.	Proving
	Determines probe adequacy.	Proving
	Supports multiple localization algorithms.	Arguing
Generality		
	Allows describing systems using different styles.	Case Studies
	Allows describing multiple problem classes.	Case Studies
	Is agnostic to system implementation technology.	Case Studies
Scalability		
	Scales to systems with a large size.	Arguing
	Scales to systems with large data volumes.	Case Studies
	Has low monitoring overhead.	Case Studies
Practicality		
	Is practical to use.	Arguing
	Allows incremental construction.	Arguing

Table 8.1: Claims established in the introduction and the detailed requirements.

images. Images are served from a separate file system storage component, shared among all web servers.

We used Znn to guide and evaluate our approach in two phases. In the first phase we described architecture-based diagnosis using Message Sequence Charts (MSCs) and used fixed time intervals for collecting traces. The results were positive and we were able to identify faults injected into the system. We then proceeded into a second phase where we replaced MSCs by a system of probes and recognizers. In the second phase we also introduced the concept of entropy described in Section 3.5.

Phase 1: Describing architecture-level behavior using Message Sequence Charts

In the first work for Znn [16] we addressed describing architecture-level behavior by identifying traces of computations. The problem is non-trivial for two reasons. First, a trace defines a finite execution. However, we are interested in systems that operate continuously, so at a system level the behavior of the system is infinite. Second, different kinds of systems embody very different kinds of computational models. For example, complex computations in a service-oriented architecture (SOA) are often defined by an orchestration script, which indicates how the various components are coordinated, and how data passes from one to another. In contrast, a system based on sensor networks may involve processing streams of sensor readings.

Our approach was based on two key ideas. The first is the idea of a transaction family. A transaction family defines a parameterized pattern of behaviors as finite computations, expressed in terms of the architectural elements (components and connectors) that are involved in that computation, and the flow of information/control between them. An instantiation of that pattern (in terms of specific architectural elements) is an individual transaction. Additionally we associated a set of properties with the components and flows. These properties indicate things like the time that a flow event happened or the load on a server. Finally, a transaction family includes a boolean function that determines whether a given transaction has succeeded.

The second idea is to associate these transaction families with architectural styles. An architectural style describes the types of elements and their possible legal associations in a system, which allows architectural patterns referring to those types to be defined. Several transaction families can be defined for each architectural style, each representing a different pattern of computation. Note that the transaction families need not cover all of the behaviors of systems in the family – only the ones that are of interest to diagnosis. However, by defining transaction families at the architectural style level, we can immediately reuse diagnosis systems for different systems. And although a different technology may be required to place probes (techniques for probing C programs are different from those for Java programs), both the diagnosis system and its configuration are fully reusable.

Phase 1: Detecting traces with Petri Nets

Given a way to specify transactions, we now need a way to observe them in a running system and then use those observations to carry out fault diagnosis. Figure 8.2 shows the process that

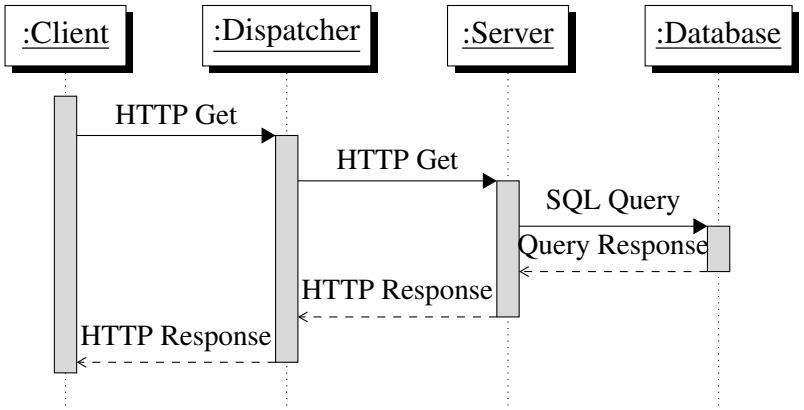


Figure 8.1: Example Message Sequence Chart for a HTTP Request Transaction.

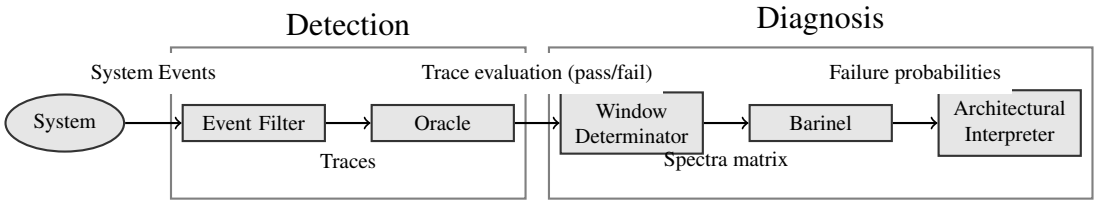


Figure 8.2: The Architecture of our Experimental Framework.

we used to do this.

To detect transactions, we adapted earlier work in architecture-based monitoring [79]. First, a system is instrumented so that it can be monitored at runtime. Monitored events are placed on an "event bus" where they can be consumed by the detection phase of our diagnostic infrastructure. In the client-server example above, monitored events include activities like initiation of HTTP requests over the client-dispatcher connectors. To monitor a system, there are numerous mechanisms that can be used that vary in terms of the kind of behavior they detect and the kind of system they are appropriate for. For distributed systems, standard middleware and network communication infrastructure provide mechanisms to monitor communication events and their properties. For systems working on a single host, code-oriented monitoring can be used. For example, aspect-oriented techniques can weave monitoring code into an existing code base (see, for example, [79]). In this phase the choice of monitoring mechanism and the placement of relevant probes was not yet been a major focus of our efforts.

During the detection phase, events are first filtered to extract those relevant to the transaction families that are being observed. Next, events are passed to a detection machine generated from the transaction families. Specifically, adapting earlier work on DiscoTect, we monitor events as a set of concurrent state machines, modeled as Petri Nets [79]. The key idea is that behavior is tracked by moving tokens through a state machine in response to low-level events. When tokens reach certain terminal states the machine emits the set of architectural elements that were involved in the trace and an indication of the transaction type. This information is then fed to an oracle that evaluates the boolean function associated with that spectrum type on the detected spectrum. The second phase of processing is diagnosis. This is broken down into three parts. The first part is window determination. In this case study, we determined the window by ex-

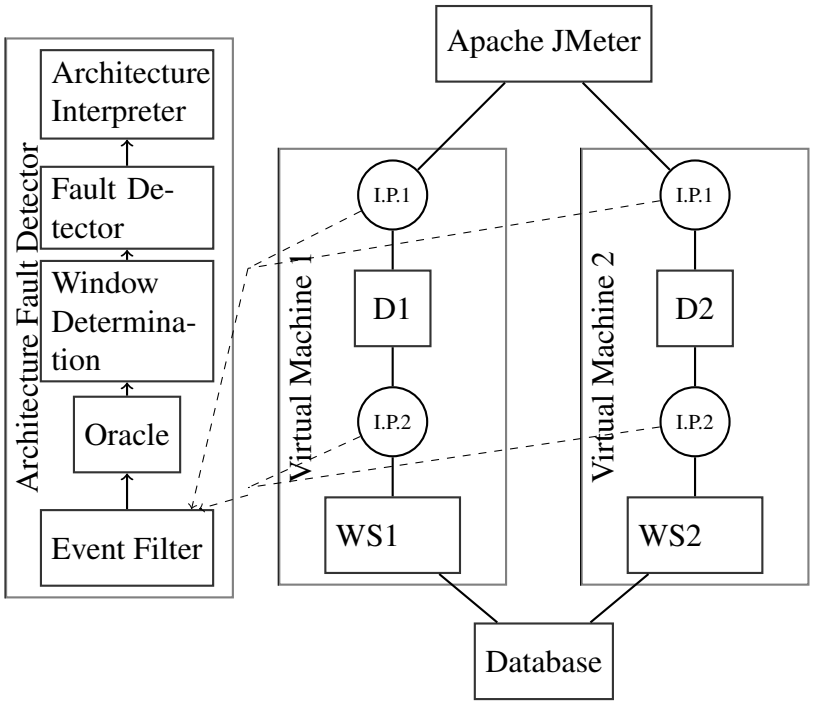


Figure 8.3: Evaluation Experiment Setup.

perimentation. Finally the results are passed to a Report Generator, which outputs the results of the Spectrum-Based Multiple Fault Localization (SFML) analysis: a list with sets of failed components and their associated probabilities.

Phase 1: Evaluation

To evaluate the approach we conducted experiments on a system similar to the example in Chapter 2. In particular, we investigated the hypothesis that the technique could accurately pinpoint problems in a system exhibiting (a) variability in the number of components; (b) distributed system structures that involve realistic, off-the-shelf communication infrastructure and componentry; (c) the presence of transient faults, where failure is based on systemic attributes like end-to-end performance; and (d) faults that might involve more than one component. The combination of these properties yields a system that would be challenging to diagnose given current technology.

One class of problems that fits this criteria are intermittent multi-component faults with additional noise. These problems arise with faulty network connections or application errors that occur only with specific combinations of input data. With these kinds of problems, a fault occurs only sometimes and is generally associated with a specific path on the system. However, other intermittent faults may occur less often due to other reasons, generating additional noise in the spectra. We want to be able to separate out the real errors from the noise.

To create this experiment, we recreated an environment similar to the one Figure 8.3. In this system, two virtual machines (simulating two servers) run two web servers and dispatchers.

	WS1	**WS2**	**Total**
D1	85/31	129/5	214/36
D2	117/5	122/0	239/5
Total	**202/36**	**251/5**	**451/41**

Table 8.2: Number of success/fail spectra for each combination of dispatcher and web server.

Time Window	**Succ./Fail.**	**Diagnosis**
0-10s	30/2	D1 : 84%,W S1 : 16%
0-20s	119/8	W S1 : 100%
0-30s	201/16	D1,W S1 : 99%, D1, D2 : 1%

Table 8.3: Time evolution of results of failure diagnosis.

The dispatchers choose which web server to send requests to using a round-robin algorithm. An external multi-threaded load test program, Apache JMeter, generates requests on both virtual machines simulating clients accessing the system. A trace family is defined for this system: a standard request in which the client performs a request to a dispatcher which forwards it to a web server. Two interception points, or probes, were placed in each machine, one before the request arrives at each dispatcher and one between the dispatchers and the web servers. These interception points (custom-developed based on the pygmy HTTP server [25]) add a specific header to the HTTP request to allow tracking the transaction and report to an event bus all events with the component name.

The fault detector receives events from the event bus and uses a Petri Net (PN) to determine to what family the transaction belongs, as previously discussed. The PN used to identify the transaction family is the one in Figure 5. Transitions on the PN are enabled when the corresponding events arrive. A transaction in the PN is initialized with a token in the `START` place and ends when a token arrives at the `DONE:Standard` place. The oracle considers a transaction to be a success if the time elapsed between the request and response is less than 2.5 seconds. This is representative of systems in which response time is a measure of success – systems that do not exhibit easier-to-detect fail-stop failures.

To simulate network delay (or server processing delay), we added a random delay in both IP1 and IP2 of the first virtual machine. This means that 25% of all requests receive an added time delay that ensures some of the time they will fail. This generates a hard-to-find problem, namely an intermittent failure on one of the paths: the one containing the dispatcher 1 (D1) and web server 1 (WS1). Simultaneously it adds a small (but non-zero) failure probability on both the D1-WS2 and D2-WS1 paths. The experimental results show that the fault detection algorithm is able to statistically separate these results and produce the correct output.

The total number of traces obtained during a run and their distribution between the various components is shown in Table 2(a). As the results show, the D1-WS1 path fails 26% of the time. The other two paths which include D1 and WS1 fail slightly less than 5% of time time and the D2-WS2 path has no failures.

Because SMFL's only input are the spectra, enough data need to be collected before the

problem can be detected. In fact, as shown in Table 8.3, the failure probabilities change over time. For example, a 20s window would have determined that WS1 was the only component responsible for the observed failures. Only after 30s is the algorithm able to indict WS1 and D1 as the components responsible for the observed failures. This means that window size needs to be carefully chosen so that the SMFL algorithm has enough information to yield accurate diagnosis.

8.1.2 Phase 2: Improving architecture-level behavior description

The work done described above was encouraging, but had some clear limitations. First, MSCs are a simple way to describe computations, but they are limited in their expressive power. We didn't use any formal language to match computations to each other and had to directly probe the events referred to in the MSC. However, directly probing the events is not always possible. For example, we might only have probes that report low-level system calls detected using Linux's *ptrace* mechanism such as `open(2)` or `bind(2)`.

In phase 2, we addressed this problem by decomposing the system behavior into a hierarchy in which detection of lower-level transactions feeds events for the detection of higher-level transactions. Figure 3.2, presented earlier, contains a partial hierarchy of events and transactions. The higher-level transactions (dynamic web request and static web requests) can be computed from the proxied requests – HTTP requests sent to the load balancer and forwarded to a web server – and database queries.

```
1  // Defining of HTTP result codes.
2  typedef int32 http_result {
3      invariant self >= 0 and self <= 599;
4  }
5
6  // Proxied request: send from client to dispatcher and
7  // then to a web server (and then back to the client).
8  computation type pxr : tc::htc {
9      result : http_result;
10 }
11
12 // Database query.
13 computation type dbq : tc::htc{}
14
15 // Static and dynamic web requests.
16 computation type dwr : tc::htc {
17     result : http_result;
18     dwr(r : http_result) {
19         result = r;
20     }
21 }
22
23 computation type swr : tc::htc {
24     result : http_result;
25     swr(r : http_result) {
26         result = r;
27     }
28 }
```

Listing 8.1: Simplified definition of dynamic and dynamic web requests.

In Listing 8.1 four simplified types of transactions (declared as `computation type`) are defined: `pxr`, a proxied request, `dbq`, a database query issued from a web server, `dwr`, a dynamic web requests and `swr`, a static web request. All these computations inherit properties

from the generic HTC (host/thread computation) in "package" `tc` (threaded computation). The threaded computation package defines the concept of a thread running in a host.

The language, which is slightly different from the final version of CALL, contains primitives akin to well-known object-oriented languages. For example, computation types are akin to classes and families to namespaces (or packages). Computation types represent events in the system, either detected by probes or fired by recognition of transactions.

Recognizing transactions from computations (events reported by probes or transactions already identified) was done through the definition of recognizers as shown, in simplified form, in Listing 8.2. The detection of these high-level transactions demonstrated the need for more expressiveness than what MSCs could give us in [16]. Static and dynamic web requests are, essentially, equal, except that dynamic web requests involve a database query whereas static web requests do not. In our recognition language, we can now express the difference in the two transactions.

```
1  // The dwr recognizer
2  recognizer dwr_rgn(r : pxr) {
3      invariant exists q : dbq | r->during_same_thread(q);
4      emit new dwr(r.result);
5  }
6
7  // The swr recognizer
8  recognizer swr_rgn(r : pxr) {
9      invariant not (exists q : dbq | r->during_same_thread(q));
10     emit new swr(r.result);
11 }
```

Listing 8.2: Identifying dynamic web requests and static web requests.

The two recognizers in Listing 8.2 identify the static and dynamic web requests from proxied requests depending on whether a database query is made during the process of the request or not. The `invariant` clause contains the first-order logic condition under which the transaction is recognized. The `emit` clause defines which transaction is identified and initializes it with the data from the lower-level computations.

Also, in Listing 8.2 we can see the other limitation of MSCs. In plain English, a proxied request is a dynamic web request when there is at least one database query performed during the request by the same thread. The *during same thread* method of the `pxr` computation type is actually defined in its super type, the `tc::htc` computation type. This method is defined as in Listing 8.3.

```
1  family tc {
2      computation type htc {
3          // ...
4          bool during_same_thread(c : htc) {
5              return same_thread(c)
6                  and start(c) >= start(this)
7                  and end(c) <= end(this);
8          }
9          // ...
10     }
11 }
```

Listing 8.3: Definition of the DURING SAME THREAD method.

The sort of reuse of connection logic provided in Listing 8.3 is not possible to achieve using MSCs. The separation of behavior into different families and structures also allows much easier

understanding and reuse. The transactions we recognize in Listing 8.2 can be used for any system that proxies requests to web servers regardless of: (1) how `pxr` and `dbq` are detected and (2) whether `dwr` and `swr` are the most high-level transactions specified or whether other transactions are defined on top of those.

Phase 2: Defining a window size

There are a number of criteria that might be used to determine this window. Based on the previous discussion and our work in [16], we decided on the following two elementary requirements:

1. It has to be large enough to produce a good diagnosis: if not enough evidence is collected, diagnosis may be inconclusive;
2. It must be as small as possible to discard out-of-date past transactions and obtain an accurate diagnosis as fast as possible: if a component fails at a certain point in time, all previous successful transactions will only reduce the confidence on the diagnosis;

In this case study, we proposed to use the concept of entropy, described in Section 3.5, to dynamically determine the window size. This avoided the experimentation required in [16] and provided better adaptation to changing faults since, after diagnosis, the past knowledge is ignored and we start collecting information to detect a new set of faults.

Phase 2: Evaluation

We evaluated the diagnostic capabilities and efficiency of the proposed approach using the Znn example described above. To illustrate how our approach can detect both functional and quality of service problems in a system, we injected five fault scenarios into Znn that manifest themselves in different ways:

- Functional failure: an image was not found in the storage.
- Functional failure: a web server has a bug and is not able to find the image to serve.
- Performance failure: a web server's response time degrades.
- Security failure: a client attempts a denial-of-service attack.
- Performance failure (correlated): a web server is slow to respond but only when requests come from a specific dispatcher.

Detecting these failures requires some correctness criteria to be defined. We define the following criteria:

- HTML response codes in the range 4xx and 5xx represent failures.
- Response times above 2 seconds represent failures.
- Client request rates over 1 request / second for at least 5 seconds represents a failure.

The experiment demonstrates that our diagnosis system is able to: (1) identify each of the faults correctly, and (2) identify multiple, correlated, faults when applicable.

We designed the recognizer as described in Section 8.1.2 by intercepting the following system calls: `socket(2)`, `bind(2)`, `accept(2)`, `close(2) read(2)`, `write(2)` and `connect(2)`. We had to track several other system calls such as `clone(2)` to keep track of the process IDs and thread IDs and we had to track other system calls that can be mapped

Time (s)	Requests	Entropy	Main candidates
1	8	0.823	d1=19.0%, fs=79.7%
2	15	0.704	d1=19.0%, fs=81.0%
3	20	0.036	fs=99.7%
4	29	0.067	fs=99.4%
5	31	0.054	fs=99.5%
6	36	0.064	fs=99.4%
7	44	0.006	fs=100%

Table 8.4: Results of scenario 1: faulty file system.

to the ones above such as `accept4(2)`, `readv(2)` and `writev(2)`. Using these, we built recognizers for the following higher-level computations shown in Figure 3.2.

We made four initial scenarios corresponding to the four fault types described above: an image is not found in storage, an image is not found by one web server (web server 1), a web server becomes slow (web server 2), and a client (client 3) acts maliciously and attempts a denial of service (DoS). The web server slowness is achieved by adding a random delay with an average of 2s (the exact limit of the allowed response time) forcing around half of the requests to fail, while allowing around half of the requests to succeed.

In all scenarios, clients 1, 3 and 4 will make requests for a web page, then request all images in it, and then will sleep for a random amount of time taken, respectively, from the distributions $N(2, 0.5)$, $N(1.75, 0.5)$ and $N(1.75, 0.5)$. This usage reflects a somewhat faster pace than a human would perform (2 seconds between pages) but speeds up convergence of the entropy. If we halve the number of requests, entropy will converge at half the speed but all other factors remain unchanged. In all scenarios except the DoS, client 2 will wait according to $N(2.5, 0.02)$. In the DoS, client 2 will wait according to $N(0.2, 0.02)$.

We aimed our diagnosis maximum entropy to 0.01 (yielding certainty over 99%). Tables 8.4, 8.5, 8.6 and 8.7 reflect the results of the first four scenarios. They contain the evolution of diagnosis over time with the total number of architecture level computations (dynamic web requests and static web requests) detected, the computed entropy and the main fault candidates. We consider $t = 0$ when the first failure occurs. It can be seen by the experimental data that the algorithm is able to correctly identify the cause of the failure in all scenarios.

Tables 8.8 and 8.9 contain the results for the fifth scenario with and without correlation fault detection, respectively. In the first case we can see that, because only the web server or the dispatcher (or both but independently) can be responsible, the system is not able to attain very low entropy values. It blames with higher probability the web server because the load balancer participates in more successful computations. If we enable correlated fault detection then we can accurately determine that the fault happens when both the dispatcher and web server are used together.

Time (s)	Requests	Entropy	Main candidates
1	14	1.211	web1=63.9%, d1=29.6%
2	18	1.226	web1=68.5%, d1=18.3%, fs=13.1%
3	20	1.078	web1=74.9%, d1=10.6%, fs=14.3%
4	29	0.647	web1=87.5%, d1=3.3%, fs=9.1%
		(rows omitted for brevity)	
18	110	0.058	web1=99.4%, d1=0.4%, fs=0.2%
19	119	0.019	web1=99.8%
20	124	0.008	web1=99.9%

Table 8.5: Results of scenario 2: faulty web server 1

Time (s)	Requests	Entropy	Main candidates
1	5	1.525	web2=63.5%, d1=12.2%, c4=12.2%, fs=12.2%
2	10	1.837	web2=42.3%, c4=24.1%, d1=24.1%, fs=1.0%
3	12	1.912	web2=35.9%, c4=25.9%, d1=25.9%, fs=12.3%
4	16	1.799	web2=43.3%, c4=31.3%, d1=14.8%, fs=10.6%
5	18	1.886	web2=34.5%, c4=34.5%, d1=18.5%, fs=12.5%
6	18	1.885	web2=34.5%, c4=34.5%, d1=18.5%, fs=12.6%
7	25	0.831	web2=76.6%, d1=22.9%
		(rows omitted for brevity)	
21	63	0.671	web2=84.1%, d1=15.5%
22	70	0.0285	web2=99.8%
23	71	0.0266	web2=99.8%
24	75	0.0257	web2=99.8%
25	79	0.006	web2=100%

Table 8.6: Results of scenario 3: web server 2 is slow

Time (s)	Requests	Entropy	Main candidates
1	25	0.000	c2=100%

Table 8.7: Results of scenario 4: client 2 tries a DoS

Time (s)	Requests	Entropy	Main candidates
1	7	1.889	web1=40.7%; fs=23.1%; c3=23.1%; d1=13.1%
2	10	1.750	web1=52.4%; fs=16.9%; d1=16.9%; c3=13.9%
3	15	1.253	web1=65.7%; d1=22.6%; fs=11.5%
4	19	1.103	web1=73.6%; d1=14.5%; fs=11.8%
5	21	1.059	web1=75.1%; d1=14.8%; fs=9.9%
6	24	0.874	web1=81.8%; fs=9.1%; d1=9.1%
7	30	0.847	web1=82.6%; d1=9.2%; fs=8.1%
8	34	0.711	web1=86.5%; fs=6.7%; d1=6.7%
9	38	0.808	web1=83.7%; d1=9.3%; fs=6.9%
10	38	0.808	web1=83.7%; d1=9.3%; fs=6.9%
11	41	0.925	web1=80.1%; d1=10.9%; fs=8.9%
12	41	0.925	web1=80.1%; d1=10.9%; fs=8.9%
13	52	0.120	web1=98.4%; d1=1.6%
13	52	0.120	web1=98.4%; d1=1.6%
		(rows omitted for brevity)	
28	115	0.132	web1=98.2%; d1=1.8%
29	120	0.126	web1=98.3%; d1=1.7%
30	123	0.145	web1=97.9%; d1=2.1%
31	127	0.130	web1=98.2%; d1=1.8%
		(rows omitted for brevity)	
54	212	0.168	web1=97.5%; d1=2.5%
55	221	0.133	web1=98.2%; d1=1.8%
56	222	0.129	web1=98.2%; d1=1.8%

Table 8.8: Results of scenario 5 without correlation detection: web server 1 is slow when requests com from dispatcher 1.

Time (s)	Requests	Entropy	Main candidates
1	3	3.168	db+c4=12.7%; db+d1=12.7%; db+web1=12.7%; db=12.7%; c4+d1=12.7%; c4+web1=12.7%; c4=12.7%
2	8	2.894	db+c4=19.3%; c4+d1=19.3%; c4+web1=19.3%; c4=19.3%; db+d1=6.5%; db+web1=3.7%; db=3.7%
3	11	3.127	db+c4=20.0%; c4+web1=20.0%; c4+d1=11.4%; c4=11.4%; db+d1=8.2%; db+web1=6.4%; db=6.4%
4	15	3.089	db+c4=20.6%; c4+web1=20.6%; c4+d1=11.7%; c4=11.7%; db+d1=8.5%; db+web1=6.6%; db=5.4%
5	20	3.011	db+c4=22.0%; c4+web1=22.0%; c4+d1=12.5%; c4=12.5%; db+web1=7.1%; db+d1=5.8 db=4.3%
6	24	1.902	c4+web1=48.8%; c4+d1=21.4%; c4=21.4%
7	27	1.430	c4+web1=73.0%; c4+d1=7.7%; c4=7.7%
8	31	2.119	d1+web1=63.2%; web1=14.8%
9	31	2.119	d1+web1=63.2%; web1=14.8%
			(rows omitted for brevity)
28	97	0.905	d1+web1=80.4%; web1=17.0%
29	100	0.818	d1+web1=83.8%; web1=13.5%
30	104	0.865	d1+web1=82.1%; web1=15.2%
			(rows omitted for brevity)
57	220	0.095	d1+web1=98.9%; web1=0.9%
58	223	0.094	d1+web1=99.0%; web1=0.9%
59	223	0.094	d1+web1=99.0%; web1=0.9%
60	227	0.103	d1+web1=98.8%; web1=1.0%
			(rows omitted for brevity)
85	328	0.024	d1+web1=99.8%; web1=0.2%
86	332	0.012	d1+web1=99.9%; web1=0.1%
87	337	0.012	d1+web1=99.9%; web1=0.1%
88	343	0.012	d1+web1=99.9%; web1=0.1%
89	344	0.012	d1+web1=99.9%; web1=0.1%
90	350	0.010	d1+web1=99.9%
91	356	0.009	d1+web1=99.9%

Table 8.9: Results of scenario 5 with correlation detection: web server 1 is slow when requests com from dispatcher 1.

The evaluation of the scenario results allow us to draw three main conclusions:

- Is possible to recognize high-level architectural transactions from lower-level events using our recognition language;
- Entropy computation provides a good way to detect when enough information has been collected for diagnosis;
- Correlated faults can be detected albeit at expense of some diagnosis time penalty.

The existence of diagnosis time penalty is theoretically predictable: because component combinations increase significantly, the number of candidates considered for fault localization increases and, consequently, more data is required. The fault localization output will, as expected, converge more slowly. However, even in this scenario, a detection with 99.9% certainty in 90s of a web server which slows half of the requests only when used with a certain dispatcher is still an encouraging result.

8.1.3 Samsung

Summary

The Samsung case study was done in collaboration with Samsung Electronics. Their semiconductor factory control system is message-based and sometimes has failures that are difficult to spot until the problem has become serious. Samsung wanted a system to detect when failures happened and quickly pinpoint the source of the failure.

The work done in collaboration with Samsung demonstrates how the approach applies in a system with significant differences from Znn:

- The system is not call-return based but event-based;
- Oracles evaluate not only correct quality attributes, such as latency, but also protocol conformance;

One of the major areas investigated in the Samsung work was throughput, and this relates to our claim of scalability. The system had to be able to cope with at least 2000 messages per second. In our terminology, this means we have 2000 low-level computations per second to be recognized into higher-level computations. Also, in this case study, Samsung defined the failure scenarios and the correctness criteria, supporting our claim that domain experts are capable of defining the oracles.

In this case study, we used CALL's architecture in its current parallel version. We measured performance and tested the expressiveness of CALL to check protocol conformance which was significantly more complex than in Znn.

This case study also shows interaction between our high-level autonomic diagnosis and lower-level recovery mechanisms. This is important as it shows that CALL can cope with systems whose architecture changes over time.

Two components in the system work in active/passive mode and if the active component fails, it is replaced by the passive one and the global computation succeeds due to the local recovery mechanism. Our autonomic diagnosis system is capable of (1) detecting the failure of the system and (2) detecting the success of the global computation correctly pinpointing the source of the failure.

The Samsung case study shows that:

- CALL can be used to define system behavior.
- Correct and incorrect behavior can be defined using CALL and the definitions are understandable by domain experts.
- CALL can correctly identify the source of failures.
- The runtime cost of CALL is low as the most performance-critical components do not need to be directly monitored.
- CALL can be used to identify failures in systemic properties.
- CALL can be used with event-based systems.
- CALL can be used to verify protocol conformance.
- CALL can be used with large data volumes.
- CALL can have low monitoring overhead.

Target System

Samsung Electronics' manufacturing system is a large-scale industrial control system responsible for manufacturing control of semiconductors. The system controls the stages of wafer manufacture, deciding to which equipment wafers are sent for processing. Furthermore, the software tracks of not only the lots being manufactured, but also the equipment used: which equipment is allocated, whether and what it is processing, what its output quality is, and so on.

The system is divided into subsystems, which perform specific tasks related to the manufacturing process. For example, the MOS (Manufacturing Operating System) subsystem controls the stages of the manufacturing process, and the ADS (Automatic Dispatch System) subsystem performs equipment scheduling, deciding which equipment is the best to perform a step in the manufacturing process.

Each of these subsystems is built by multiple concurrent processes that provide various manufacturing services, load distribution, and fault tolerance. These are connected through an event bus, which mediates event exchange amongst them. A subset of the system's architecture is abstractly depicted in Figure 8.4. Due to confidentiality reasons, specific details about system implementation cannot be reported. The total message throughput in all buses combined is around 2000 events per second.

The systems communicate with each other exchanging messages according to several predefined protocols. One such protocol is the *track-in* (`TKIN`) protocol. This protocol is used before a wafer lot is sent to another stage of processing. It determines what equipment the wafers should be sent to, performs validation operations and does some housekeeping, like ensuring that the equipment for processing the next steps are available and that no scheduling and quality constraints are violated.

Several factors make this system complex:

1. The `TKIN` protocol contains more than just message exchange between components: it also involves databases accessed through standard protocols. This means that observation of the event bus will only yield partial information.

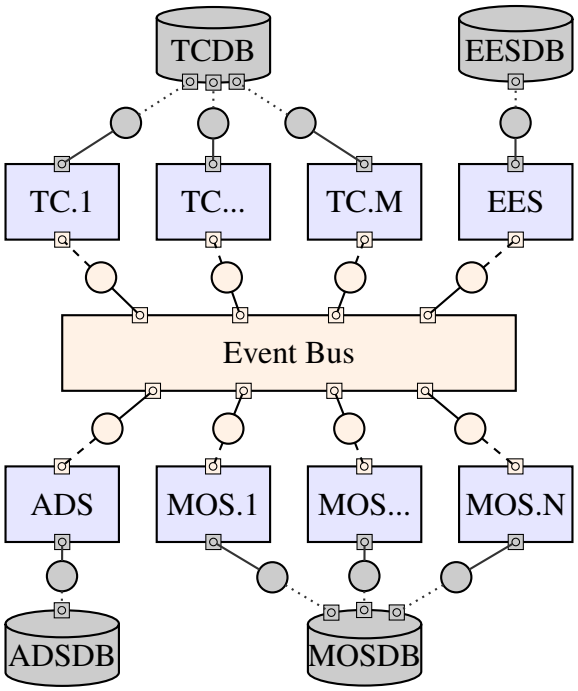


Figure 8.4: High-level architectural view of the manufacturing system at Samsung Electronics. For simplicity, only the MOS and TC systems are shown decomposed.

2. Several components of the same type exist, but only one will receive each message although it is not known which. This makes tracking messages harder as we don't know what the destination of the message will be: all possible destinations have to monitored and searched for a match.
3. Multiple TKIN protocols are occuring in parallel and they may be executed with different timings. For example, the MOS may send two BEST_EQP messages to the same ADS which returns two EQP responses whose contents have to be analyzed to identify which response matches which request.
4. More messages circulate through the bus between these components to address other protocols besides the TKIN protocol. This means that we will see spurious messages in the bus.

Operation issues While TKIN protocols are executed correctly most of the time, once in a while problems arise that cause failures in the system. These problems vary, but some of the typical problems are messages being lost (or not sent at all), messages sent too late, or unexpected messages being sent. Other problems are database performance slowdowns, which affect overall system performance.

Such problems can have a significant impact on the manufacturing process as they can lead to stalling lots and unneccesary equipment reservations. Given the sheer volume of messages being exchanged it is not possible for human operators to even realize that a problem has occurred until later when more serious consequences become visible. Also, given the independent development

of all the systems involved, it is difficult for the system developers to figure out what the problem is and where it is located.

Yet another difficulty is that these problems, although serious, are rare - and therefore are difficult to diagnose. Although the system operates with around 2000 messages exchanged per second, it generally functions correctly for many weeks in a row before any failure is detected. This sets up a scenario in which millions of observations are performed before a single failure can be identified.

Simulating the target system There are several barriers to developing a diagnosis system for systems such as the Samsung Electronics' manufacturing control systems. The criticality of such systems makes testing on the production system unacceptable; the size of the system and its need to work connected to factory equipment makes it impossible to run it in a different environment. Also, the rareness and unpredictability of the faults make testing a diagnosis system difficult.

We addressed these issues by creating a simulator of Samsung Electronics' production system and performing diagnosis in the simulator. This simulator was engineered with the help of Samsung Electronics to produce a `TKIN` protocol resembling the real one and to allow manual control of faults for testing purposes.

The `TKIN` protocol is described using a message sequence diagram in Figure 8.5. The messages in the sequence diagram correspond to types of events sent: for example, `BEST_EQP` is a request for an equipment to process a wafer lot and `EQP` is the event with the ADS's decision on the equipment to perform the process step.

The simulated system does not, naturally, control any equipment. Rather, it generates messages that conform to the protocol, both in terms of their types and also in their timing. This simulator allows us to develop test scenarios in which we arrange for specific faults to happen, check whether they are diagnosed correctly, and have a credible expectation that it can be successfully transitioned into Samsung Electronics' production system.

The simulated system is similar to the one in Figure 8.4, but it has only two `MOS` components, one event bus, and two `TC` components. This is in contrast with the real system which includes more than 20 of each type. However, because the computation performed by our individual components is much simpler than the real one, they are able to generate a much higher message rate than their real counterparts yielding a message rate closer to the real system. Consequently, our smaller number of components is actually simulating a higher number of real system components from a message exchange perspective.

The system starts multiple `TKIN` protocols at random time intervals. Components have random processing delays and a single `TKIN` protocol takes around 1 minute to complete, similar to the `TKIN` protocol in the real system. The rate at which `TKIN` protocols are initiated can be used to control the level of concurrency.

Both `MOS` components work in fault-tolerance mode: either `MOS.1` or `MOS.2` will receive messages addressed to the `MOS`, but not both. They maintain a "keep-alive" mechanism, allowing one to take over if the other fails to respond. The `TC` components work in "load-distribution" mode: each request will be forwarded to *either* `TC.1` or `TC.2`.

Implementing fault tolerance in our simulator was done by creating a synthetic component (`MOS.S`) that acts like the `MOS` for all other components (except `MOS.1` to `MOS.2`). It will

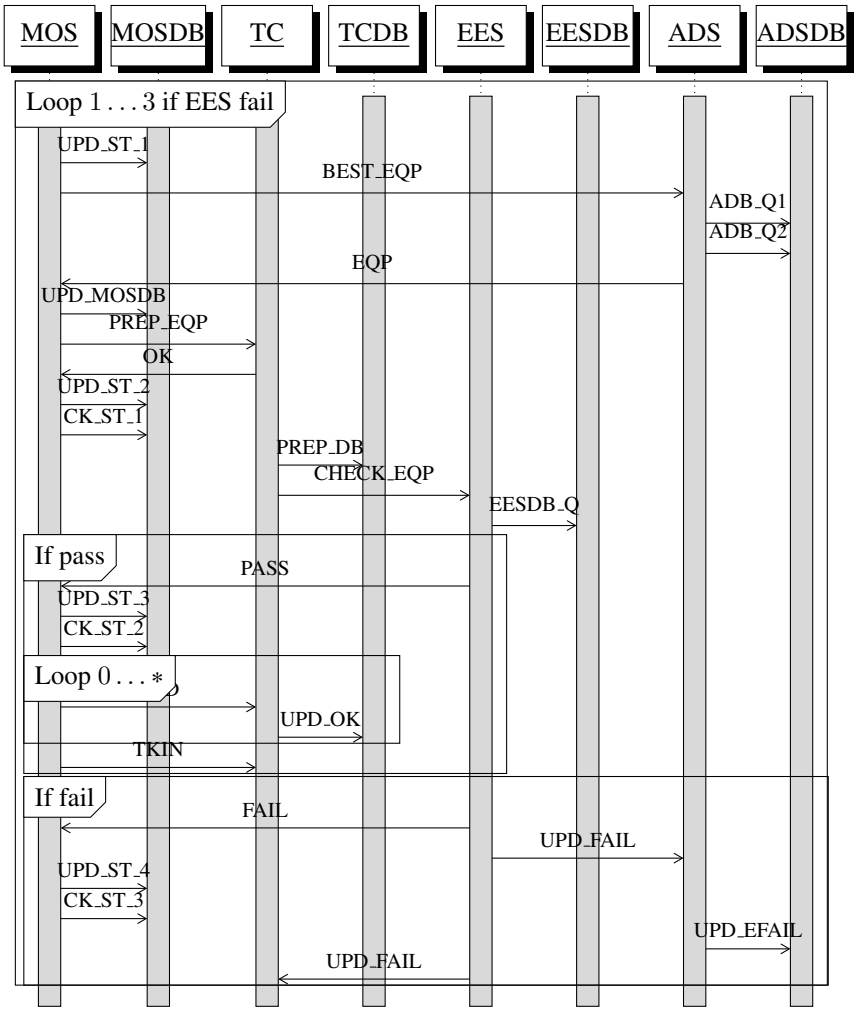


Figure 8.5: Simulated TKIN protocol.

forward through the event bus any message to either `MOS.1` or `MOS.2`, depending on which component is active. Load distribution is implemented similarly using a synthetic `TC.S` component. This mimicks the way the event bus system used at Samsung handles load balancing and fault tolerance.

Although we did not have access to the actual factory control software, we were careful to work with Samsung Electronics engineers to ensure that the simulator we developed was faithful to the real system in the following ways:

- event timings were designed to produce delays close to the real system;
- the protocol was designed according to Samsung Electronics' specifications;
- the faults that were injected were typical of real problems experienced by Samsung Electronics and which are hard to locate in the real system.

We are therefore reasonably confident that the results reported in this case study will apply when Samsung Electronics transitions our approach to the real system.

Application of the Framework

Applying our approach to the system at Samsung Electronics required defining the behavior of the system in terms of CALL. The main lines of our approach were:

- Each bus message was recognized to produce several messages: a message being *sent* by components and one message (or more, depending on the destinations) being *received* at components.
- The receive/send message pairs that contain call/return behavior, were recognized as a single computation.

  For example, the BEST_EQP received at the ADS is paired with the EQP sent by the ADS to produce a ads_eqp computation.
- Each pair also produced recognizers that detect behavior failures on timeouts.

  For example, ads_eqp_no_response recognizer recognizes a BEST_EQP for which *no* EQP exists.
- Sequential computations were then bound together in a single computation that represents a sequential flow. We ended up with three sequential flows: eqp_prep which corresponds to the flow until the EES decides on a pass or fail, a test_pass which corresponds to the "pass" optional section in the protocol flow and a test_fail which corresponds to the "fail" optional section in the protocol flow.

  For example, the ads_eqp is matched to the mos_start (recognized from a BEST_EQP detected at the MOS) to yield a mos_ads_eqp. The mos_ads_eqp is matched with a mos_prep (recognized as a EQP/PREP_EQP pair at the MOS) to yield a mos_eqp_prep and so on.
- The alternative flow was modeled using four recognizers: one that recognizes a eqp_prep and a test_pass into a eqp_pass, one that recognizes a eqp_prep and a test_fail into a eqp_fail, one that ensures either test_pass or test_fail, and one that recognizes a single_run from either an eqp_pass or an eqp_fail.
- Loops were modeled using recursion and loop limitation by placing a counter in the computation.

  For example, the tkin computation type has an attribute m_count that specifies the run count. A tkin can be recognized from another tkin and a single_run. The newly generated tkin has a m_count which is one value higher the the previous tkin.

It is worth discussing more deeply the matching of call / return patterns within the TKIN protocol. If the contract for the ADS is, as is the case, to *always* reply with an EQP to a BEST_EQP request, then receiving an EQP and not replying with a BEST_EQP is a failure regardless of whether there is any other component involved. However, if the contract for the ADS were to only reply to *some* requests (for example, invalid requests would not have a response), then the pair BEST_EQP/EQP would not be enough to pinpoint a failure in the ADS component and the call / return pattern could not be applied.

The previous discussion hints at two important aspects of our approach: (1) because behavior is formally modeled, it reduces ambiguity in system specifications, and (2) the small pieces of the protocols follow patterns that are reusable across multiple systems, while the large, complete

protocols generally do not.

Metrics

To evaluate how well the diagnosis algorithm performs, we need to identify a set of appropriate metrics. There are two main areas of metrics that are applicable in domains sharing similarities with ours: metrics used for evaluation of classifiers and metrics used for evaluation of fault localization algorithms.

Our algorithm has similarities with classifiers in machine learning: it will output a classification of which components are faulty and which are not. In that sense, the classic metrics of precision and recall, or one of its variants, seem to be relevant. However, those statistics are applicable only in binary cases [61] and our classifier outputs a *probabilistically ranked* classification. Many components – maybe even all – may show up as outputs but their *probability*, or *rank* is relevant: if the faulty component has probability of 0.99 and all the others have probability $\frac{0.01}{n-1}$ where n is the total number of components, this is generally seen as a good classification although it has a low value of precision. In the presence of few faulty components, recall will always tend to the high or low spectrum. These metrics are, therefore, not applicable.

The fault localization literature uses ranking for evaluation: the real probability attributed to diagnosis is only relevant in comparison with the others. The rationale behind this metric is that developers will tend, in order to correct the faults, to inspect components by rank order and, therefore, the lower the rank of the faulty component, the more effort will be wasted. This metric is applicable to evaluate run-time diagnosis.

However, at runtime, timing (performance) also becomes an important issue. If at design time, the time taken for diagnosis is not relevant, at runtime it may be critical for autonomic recovery. Also, at runtime, many diagnoses can be produced for the same system and they may yield different values. Therefore, we introduce two metrics:

1. **Failure identification time (FIT)** that measures the time between when the fault is activated and when the diagnosis system presents some information that *something* is wrong;
2. **Diagnosis stabilization time (DST)** that measures the time between failure identification and diagnosis stabilization: when the system decides on which diagnosis to consider final.

While for some systems – like the one presented in this case study – it is the sum of both metrics that is relevant, in other circumstances each metric may have different impacts: e.g., if a breach in a high-security system is detected (failure identification time) we may want to disconnect the system from the network immediately even before computing which part of the system was breached (diagnosis stabilization time).

Evaluation

We evaluated the system by defining the computations and identifying a set of *scenarios* in which the system would fail in some predicted way. These scenarios were designed to replicate the types of real problems encountered at Samsung Electronics.

1. `EES` sends a `FAIL` message and, 3s later, sends the `PASS` message. Because the `MOS` subsystem responds to both events, this starts two parallel flows: one for the successful

evaluation, which ends with a TKIN, and another for the unsuccessful evaluation, which ends with a retry of the whole process.

2. TC database is too slow to respond.
3. TC sends CHECK_EQP to ADS instead of EES.
4. Active MOS fails and is later replaced by passive MOS.
5. MOS retries four or more times.
6. ADS does not issue ADB_Q1.
7. The ADS database is too slow to respond.

Evaluation addressed the two main goals stated in the introduction:

1) diagnostic accuracy, and
2) diagnostic performance.

Based on our description above, we measured the accuracy of the system as the rank of the faulty component in the diagnosis and we measured performance by computing the failure identification time and diagnosis stabilization time.

We also computed another measure, not directly related to the diagnosis system, but of relevance to Samsung Electronics: an estimate of the maximum throughput of the system. Since every recognizer can be run in a different system, the maximum throughput of the system is limited by the amount of time the slowest recognizer takes to perform its evaluation.

Accuracy The average result of measuring accuracy in the 7 scenarios is presented in Table 8.10. These results show that the system was able to correctly detect a failure in all scenarios except scenario 6. Because databases are a source of performance bottlenecks at Samsung Electronics, it was not possible to determine communication between ADS and ADSDB. Consequently, the diagnosis system has no way of distinguishing between the slowness of ADS and ADSDB; each component is equally likely to be the cause of the slowness. Interestingly, we are able to distinguish the slowness of TCDB. Because the two TC components connect to the same database, slowness in both increases the probability of the fault being localized in the database itself. The fact that some faults can be accurately pinpointed and some not was discussed formally in [19].

Performance Performance metrics evaluated in all scenarios are shown in Table 8.11. For each evaluation scenario we ran the scenario 10 times and collected the two performance metrics discussed above. The fault localizer outputs the results regularly (once every second).

As noted earlier, the FIT is the time between the fault being activated and a diagnosis being produced that includes an identification of a failure. If this diagnosis result already included the failed component ranked in the first position, then the diagnosis stabilization time (DST) is 0. In several scenarios, a later diagnosis never placed another component in first position, meaning that most scenarios have an instantaneous DST. In scenario 2 however, because TCDB was not instrumented, detecting that it was the source of the failure required more data and, therefore, DST is greater than 0.

Scenario	Component†	Rank‡
1: EES sends FAIL and PASS	EES	1
2: TC database is too slow	TCDB	1
3: TC will send message to ADS instead of EES	TC	1
4: MOS fails and is replaced by standby	MOS.1	1
5: MOS will retry 4 times	MOS.1*	1
6: ADS will not issue ADS_Q1	-**	-**
7: ADS database is too slow	ADSDB	2***

Table 8.10: Evaluation of performance metrics in the scenarios.

† Component where the fault was injected.

‡ Average rank among 10 scenarios.

* MOS.1 was the active MOS.

** No failure was detected by the diagnosis system in this scenario.

*** ADS and ADSDB were both ranked with an equal probability of 0.5 in all 10 scenarios. We count as 2 as a developer / system maintainer may want to inspect the ADS before the database. We took the most conservative approach.

Scenario	avg FIT†	std FIT††	Min FIT†††	avg DST‡	std DST ‡‡
1: EES sends FAIL and PASS	5,506	319	3,000	0	0
2: TC database is too slow	13,506	3,212	5,000-15,000	436	432
3: TC will send message to ADS instead of EES	17,708	309	15,000	0	0
4: MOS fails and is replaced by standby	15,197	1,505	10,000-15,000	0	0
5: MOS will retry 4 times	2,861	0,343	0	0	0
6: ADS will not issue ADS_Q1	-*	-*	-*	-*	-*
7: ADS database is too slow	10,204	1,131	5,000-10,000	0	0

Table 8.11: Evaluation of performance metrics in the scenarios. Results in milliseconds.

† Average failure identification time. †† Standard deviation of failure identification time.

††† Theoretical minimum failure identification time. ‡ Average diagnosis localization time.

‡‡ Standard deviation of diagnosis localization time. * No failure was detected by the diagnosis system in this scenario.

The minimum FIT column is added for reference. It is the minimum theoretically possible FIT that would detect the failure. In scenario 1, for example, the FAIL message is sent 3s after the PASS message so, before that, no failure can be detected although the invalid code has already been started. In some examples, like this one, we could have deducted the minimum FIT from the FIT but due to scenarios in which the timing is not so predictable (like introduced database slowness in scenarios 2 and 7) this would not be consistent.

Lessons Learned

The application of our diagnosis framework to the system at Samsung Electronics provided us with confirmation that several of our assumptions appear to be supported in an industrial setting and also provided us with insight into some other areas.

Decomposing computations is critical to the success of the diagnosis. As a result of the decomposition, failures can be identified not only in the high-level computations but, sometimes, also in lower level computations. This result allows diagnosis sometimes to be performed in a much smaller set of components yielding a much faster response and a higher accuracy. For example, a lack of EQP response allows inferring immediately a fault in the ADS component due to the request / response low-level computation model.

Even with strict protocols, there are scenarios where reasoning-based diagnosis analysis is of added value. The previously mentioned decomposition allows local detection and identification of the source of the failure in some cases. This is consistent with industrial practice in which many instances of this reasoning – like heartbeats or timeouts – are used. However, in some cases this is not sufficient. As seen, for example, in scenario 2 where the TC database slows down, more complex reasoning may be required to accurately pinpoint the source of the failure.

Behavior specifications can be built up using patterns that are reusable across systems. The computations that comprise the behavior specification we build are defined on top of smaller

patterns of well-known computation styles like call/return, alternative flows or loops. Therefore, although different systems may need to specify their own idiosyncratic high-level computations, a significant number of specification blocks may be reused across systems.

Domain experts are able to provide correctness specifications for high-level computations. Domain experts would not be able to explicitly state that within 15 seconds after a `BEST_EQP` message had been seen in the event bus, an `EQP` message from the same ADS to the same MOS with the same `lot_id` field should be seen in the event. This reasoning involves a great deal of low-level detail. However, domain experts *were* able to state that the ADS should respond within 15 seconds of a request for an equipment. This means high-level computations have an abstraction level that matches the domain experts' understanding of the system and are, therefore, a good level to write correctness specifications.

Instrumentation design must consider a trade-off between system performance, diagnostic accuracy and diagnostic performance. The limitation on the observability of some events creates uncertainties which, in some cases like scenario 2 were solved by the reasoning-based analysis with a performance penalty. But in other cases, like scenario 7, this was not possible. However, instrumenting the connection between the ADS and its database was not possible because it was considered by Samsung to introduce an unacceptable performance penalty.

Architecture-level autonomic diagnosis systems can work in the presence of local recovery mechanisms. As shown in this case study, the failure of the active component in an active/-passive cluster that triggers a switch to the passive component is monitored by our autonomic diagnosis. The faulty component is correctly pinpointed even if the higher-level computation succeeds.

8.2 Requirements Validation

Allows describing system behavior using a declarative approach. It can be seen in both case studies the system behavior described using behavior declaration. In Znn we described what an HTTP call is by declaring how it is constructed. In the Samsung case study, we declare the structure of the TKIN protocol.

Allows describing what is correct and incorrect behavior also using a declarative approach. This requirement is validated much like the former one. It can be seen in both Znn and Samsung case studies.

Provides mapping from failures to their sources. Both case studies show this feature of CALL. In Znn, CALL pinpoints which servers have failed and in Samsung which components are not responding. The mapping is not always perfect nor always accurate. Both case studies show this behavior. However, as was shown in the use cases (and formal proof discussed in their publications), the actual source of failure is always identified. The lack of accuracy under some circumstances means that sometimes other possible sources of failures are identified.

Allows easy code reuse between systems. CALL makes extensive use of libraries and inheritance. By using widely-known reusability concepts from popular languages such as C++ or Java,

CALL guarantees that its own constructs can be reused between projects. Both case studies show these features by reusing definitions of patterns of computations. For example call/return communication was used in both systems and was provided as a library that can be further reused.

Has a low runtime probing cost. Having a low runtime probing cost implies that it is possible to reason about unobserved behavior and probe deployment, thus reducing probing requirements. This is shown clearly in both case studies.

For example, in the Znn case study it was shown that is possible to reason about connector health without explicitly monitoring the connectors. In the Samsung case study it was possible to reason about the health of the databases without probing them.

Also, work made in [19] shows that it is possible to not probe components in situations where collected traces allow inferring their presence in the computation.

Provides support for systemic system properties (*i.e.*, quality attributes). Both case studies show support for systemic system properties. The Samsung case study shows support for performance and dependability. The Znn case study shows support for performance.

Guarantees system coverage. System coverage is guaranteed through coverage analysis based on the results of [19].

Systematically finds and orders failure cases. Guarantees of fault identification are the ones provided by the choice of SFL algorithm. Barinel [3] provides several guarantees, in particular that, in a single-fault scenario, the actual fault is never ranked below any other candidate faults, although it may rank equally.

Determines probe adequacy. Probe adequacy is also guaranteed through *consistency analysis* as described in Section 3.6.

Supports multiple localization algorithms. As shown in Section 3.1, CALL collects architecture-level computations with correctness evaluation and feeds them to the fault localizer.

In the implementation for this thesis, the Fault Localizer was implemented using a specific SFL algorithm: *barinel* [2]. However, different implementations of the Fault Localizer can be used. Two key ideas illustrate the flexibility (and its limits) of changing localization algorithms:

- *The whole Fault Localizer component can be replaced.* Oracles publish computations with their evaluation on an event bus and the Fault Localizer reads those computations from the event bus. Therefore, a different Fault Localizer can be developed and plugged into the event bus.
- *The SFL algorithm in the Fault Localizer can be replaced.* If an SFL-style approach is to be used and the same concept of entropy used to determine the trace collection window is to be retained, then a different algorithm other than barinel can be used. Barinel is used as an external library and is easy to replace. For example, Tarantula [42] can be used instead.

Allows describing systems using different styles. The two case studies show systems with two different styles whose behavior is defined in CALL: Znn is a call/return based system and Samsung is an event-based one. This shows that CALL can handle systems with at least two different styles. But through analysis of CALL's syntax and semantics we can see that there are no style-specific constructs or concepts. Both case studies show an application of CALL's capabilities.

Allows describing multiple problem classes. Support for multiple problem classes is deducible from the oracle classification capability. The Samsung case study provides a concrete example of simultaneously verifying functional requirements, performance and dependability.

The dependability quality attribute in the Samsung case study is particularly relevant. The system has a standard self-recovery mechanism built-in: there is a standby MOS that will replace the active MOS if it fails. While CALL can be used to describe MOS behavior and identify when it fails, it can also – as done in the case study – be used to identify that the self-recovery mechanism is operating correctly and the standby MOS will take over the active MOS if the latter fails.

Is agnostic to system implementation technology. Both case studies have very different technologies: Znn is a standard LAMP (Linux, Apache, PHP, MySQL) stack and Samsung is standard Java.

The independence of the monitored system's implementation technology is ensured by using an event bus to report events to CALL. Of course, probes themselves need to be able to communicate with the system and are specific for each technology. For example, in the Samsung case study, we connected to the system's own event bus to probe events. In Znn we used OS-level system call tracing to identify calls.

But, with the exception of the actual implementation of the probes, once the computations are detected and sent to CALL, no dependency on the system's implementation technology exists.

Scales to systems with a large size. Given the architecture of CALL, the actual system size is mostly irrelevant since it is the number of collected computations that affects the performance of CALL. It is the volume of data that is relevant for CALL and that is discussed below.

Scales to systems with large data volumes. The Samsung case study shows scalability, achieving almost half of the scale of Samsung's production system in a low-end laptop. Further scalability is guaranteed by the distributed architecture of CALL, limited by CALL's event bus' capability of receiving and routing events.

The scalability of the Fault Localizer is dependent on the algorithm used and barinel has shown to perform very fast.

Has low monitoring overhead. Both case studies show a low monitoring overhead, but the Samsung one shows this very clearly, with a dual-core 2.26GHz Intel P8400 CPU can handle around 900 events / second.

Is practical to use. See Section 8.3

Allows incremental construction. Incremental construction can be done in CALL in three ways: bottom-up, top-down or scope-wise. All three can be used in independent iterations or together in the same iteration.

Bottom-up incrementality represents increasing the abstraction of computations in different iterations. This is commonly used when building the system incrementally and was used heavily in the case studies. For example, considering the example in Figure 3.2, in a first iteration one could recognize the "HTTP request with IP" computation only and place an oracle that evaluates whether this computation corresponds to correct or incorrect behavior. Then, in a second iteration, a recognizer that would match a "HTTP request with IP" with an "HTTP response" would be added and an oracle that evaluates that computation added and so on.

Top-down incrementally is used in conjunction with probe placement and is particularly useful to do incremental diagnosis. For example, still with the example of Figure 3.2, one could detect a proxied request by just putting an HTTP proxy between clients and dispatchers in the Znn use case. However, diagnosis would be vague as no internal server components could be diagnosed independently.

However, if a failure is detected, more probes could be added and the other recognizers be placed so a more detailed diagnosis can be performed.

Scope-wise incrementality is used when only part of the system is monitored and, as iterations progress, further parts of the system are monitored. This allows for a divide-and-conquer approach to building the diagnosis system.

Regardless of how incrementality is used, it is not a feature of the language or the runtime environment, but rather enabled by the language: recognizers and oracles can be added independently of other recognizers or oracles already present, allowing bottom-up and scope incrementality; probes can be added independently of other probes and recognizers can replace probes independently of the rest of the specification, allowing top-down incrementality.

8.3 Validating Practicality

Practicality is validated by showing that CALL does not introduce any concepts that would be unfamiliar to a standard software engineer – specifically, those found in standard programming languages and basic set theory and logic. This means that, apart from the initial learning curve required to learn the syntax and how to setup CALL-RE, no new fundamental concepts are required to be learned.

Table 8.12 contains the main concepts in CALL. Many trivial concepts widely used in computer software such as parameters, code comments or constants are not included. These are used in CALL, as in mainstream computer languages.

The next four tables, Table 8.13, Table 8.14, Table 8.15 and Table 8.16 map, when possible, the CALL concepts to those in three major programming languages (Java, C++ and SQL) and the mathematical concepts in first-order predicate logic.

Concept	Description
Primitive types	Primitive types are extensively used in CALL for multiple purposes: defining data in computations, temporary variables used in recognizers and oracles, etc.
Collections	Collections (sets, lists, bags, etc.) are used to hold data just like primitive types.
Complex data types	Complex data types such as structures, typedefs and classes are used to represent data, just like primitive types or collections, but are also used to represent computation data.
Data invariants	Invariants are used in complex data types to ensure that they can only be in a valid state.
Immutability	Many values in CALL are immutable. This facilitates implementation of the distributed model and improves its performance.
Operations	Operations are used in some complex data types. They provide algorithms associated with the types.
Inheritance	Inheritance is widely used in complex data types in CALL.
Namespaces	Data types in CALL are organized in namespaces to avoid naming clashes and providing reuse.
Scopes	Variables and other concepts are scoped meaning they are only accessible in part of the code.
Libraries	Libraries are collections of CALL files with definitions of data types, recognizers and oracles. They a key element of reuse.
Computation	Computations represent observed or inferred operations made by a system. They are the fundamental reasoning element of CALL.
Stream	Streams are time-based partially ordered collections of computations identified by probes or inferred by recognizers.
Event bus	An event bus sits at the core of CALL-RE's implementation. It provides distribution of computations to recognizers and oracles.
Recognizer	Recognizers are the inference mechanism in CALL. The identify higher-level computations from lower-level computations.
Oracle	Oracles are classifiers that identify computations as either correct or incorrect.
Binding steams	When setting up CALL-RE, all streams are bound to probes, recognizers and oracles to generate a runtime graph.

Table 8.12: Main concepts used in CALL.

Concept	Description
Primitive types	Same concept, although java's primitive types are different.
Collections	Same concept as java's collections. Java does have different collections and no specific syntax to use them. However, very common libraries such as Google's Guava [36] add most of the collections CALL has.
Complex data types	Java has classes with the same meaning as, CALL, but does not support typedefs. Java also does not have structures, but it does have a convention for Java Beans that perform an equivalent role.
Data invariants	No mapping.
Immutability	Java has limited support for immutability through the `final` construct.
Operations	Java methods.
Inheritance	Java has single class inheritance and multiple interface inheritance. The mechanism is less general, but essentially the same.
Namespaces	Java packages.
Scopes	Same concept.
Libraries	Same concept.
Computation	No mapping.
Stream	No mapping.
Event bus	The Java runtime itself has no concept of an event bus. However, many widely used implementations exist.
Recognizer	No mapping.
Oracle	Java predicates.
Binding steams	No mapping.

Table 8.13: Mapping CALL concepts to java.

Concept	Description
Primitive types	Same concept, although the names of the types are different. C++ has more expressive primitive types.
Collections	C++ has no collections in the language although the official STL [23] has several containers. The widely used boost library [24] also provides more containers.
Complex data types	C++ has classes, structures and typedefs. C++'s typedefs of primitive types cannot have methods.
Data invariants	No mapping.
Immutability	C++ `const` keyword.
Operations	C++ methods.
Inheritance	Same concept, with C++ being more permissive than CALL.
Namespaces	Same concept.
Scopes	Same concept.
Libraries	Same concept.
Computation	No mapping.
Stream	No mapping.
Event bus	C++ has no concept of an event bus, although many implementations exist and there are standards in many operating systems.
Recognizer	No mapping.
Oracle	C++ predicates.
Binding steams	No mapping.

Table 8.14: Mapping CALL concepts to C++.

Table 8.17 contains a summary of all four previous tables.

Mapping between CALL and Java Unlike CALL, Java has little built-in support for collections. Java's standard language supports arrays and most of the collections are built from more elementary primitives. Java has built-in support for some concepts just as `Iterable` to allow language constructs like the `for-each` loop. However, the most important collections in CALL are available either in standard Java or in well known libraries.

Java has no concept of invariants, in itself, although constraints on data values can be implemented manually in code. It also has no built-in support for immutability of complex types, although encapsulation and the use of `final` can be used to implement immutable types.

Java has no concept of computation or streams of computations. These could be implemented with the language's primitives, of course.

Java also has no concept of a recognizer and implementation of a recognizer is Java is not straightforward.

Mapping between CALL and C++ C++ contains most of the features available in CALL. The important exceptions are computations and streams (both easily implemented in C++) and

Concept	Description
Primitive types	Same concept, although different primitive types.
Collections	No mapping in general, although some vendor-specific extensions, such as Oracle's nested tables to allow for a similar concept, although much more limited.
Complex data types	Tables are similar to structures. No mapping for classes exist. Most SQL implementations do allow some definition of user-defined data types that are similar to typedefs.
Data invariants	Constraints.
Immutability	No mapping.
Operations	SQL functions are a similar concept, although not associated to tables and, therefore, not equivalent.
Inheritance	No mapping.
Namespaces	Schemas.
Scopes	No mapping.
Libraries	No mapping.
Computation	Row.
Stream	Table.
Event bus	SQL has triggers that allow for a simple, but limited, version of an event bus.
Recognizer	View.
Oracle	Predicate.
Binding steams	Joins.

Table 8.15: Mapping CALL concepts to SQL.

recognizers. Recognizers are not easy to implement in C++.

Mapping between CALL and SQL SQL is a query language designed to specify patterns of data. SQL has very limited capabilities to describe complex data structures as its foundation is around tables and columns. It is possible to describe complex data structures in SQL by use of multiple tables, but that is not equivalent to describing complex data structures in CALL.

Some vendor-specific extensions allow more complex data structures to be held in tables. Some object-relational databases allow Java classes be defined as columns, but these are not standard nor commonly used. This means that "simple" concepts such as a class containing a list of numbers is not mappable in SQL.

SQL has no concept of data immutability, although database management systems do have concepts such as access permissions and read-only data. We consider that these concepts are different and not mappable to CALL, even if they can be used to attain the same goal.

SQL also has no concepts of table inheritance or any concept of table libraries. Specific vendors have extensions to the language to enable some of these features, but they are uncommon.

SQL has no event bus and, while triggers can be used to perform actions when data is inserted into a table, they are hardly a replacement for an event bus, especially performance-wise.

Mapping between CALL and First-Order Predicate Logic There are many variants of FOPL and many implementations of theorem provers provide their own extensions. Also, FOPL-like specifications like Z may allow for many features like schemas and functions, but base FOPL has no such concepts.

Recognizers and oracles do not exist in FOPL itself and cannot be directly implemented as the concept of computation does not exist. But matching values based on predicates and logical statements is a core feature of FOPL and so a good part of what is a recognizer and an oracle has direct mapping to FOPL.

As can be seen from Table 8.17, all main concepts in CALL are known to any software engineer who knows Java, C++, SQL and FOPL.

Concept	Description
Primitive types	FOPL does not have an explicit concept of primitive types, although all given sets can be seen as primitive types.
Collections	FOPL has direct support for sets and tuples, but no direct mapping for other collections such as maps or lists. However, those structures are easily built from the first ones.
Complex data types	FOPL has no concept of structures or classes, although predicates can be used to define properties of items.
Data invariants	FOPL's axioms can express invariants.
Immutability	FOPL does not have a concept of mutable value, so all values are immutable.
Operations	No mapping.
Inheritance	No mapping.
Namespaces	No mapping.
Scopes	No mapping.
Libraries	FOPL itself does not have libraries, but many of the places where it is used like theorem provers support it.
Computation	No mapping.
Stream	No mapping.
Event bus	No mapping.
Recognizer	No mapping, although the concept of defining an invariant covering more than one set is widely used in FOPL.
Oracle	No mapping, although the concept of a predicate that classifies values as either true or false is widely used.
Binding steams	No mapping.

Table 8.16: Mapping CALL concepts to first-order predicate logic.

Concept	**Java**	**C++**	**SQL**	**FOPL**
Primitive types	Y	Y	Y	P
Collections	Y	Y	N	Y
Complex data types	P	Y	P	N
Data invariants	N	N	Y	Y
Immutability	N	Y	N	Y
Operations	Y	Y	N	N
Inheritance	Y	Y	N	N
Namespaces	Y	Y	Y	N
Scopes	Y	Y	N	N
Libraries	Y	Y	N	Y
Computation	N	N	Y	N
Stream	N	N	Y	N
Event bus	Y	Y	Y	N
Recognizer	N	N	Y	P
Oracle	Y	Y	Y	P
Binding steams	N	N	Y	N

Y means an equal or similar concept exists in the target language. P means a concept that has similarities exists and therefore the concept is considered to be partially covered by the target language. N means no similar concept exists.

Table 8.17: Summary of mapping CALL concepts.

Chapter 9

Discussion

We have presented a framework that performs failure detection and localization by analyzing patterns of computations. Probed computations are usually low-level and the framework provides a way to abstract them into higher-level computations, which can then be evaluated for correctness and used to pinpoint failures in a running system.

We presented a language, CALL, through which computations are declared and rules for abstraction (recognizers) and classification (oracles) are defined. We also showed how failures could be localized based on traces of both successful and failed computations using techniques such as spectrum-based localization.

In this section we will discuss limitations and consequences of this work. We will start by discussing, in Section 9.1, what kind of failures can and cannot be detected.

One of the main contributions of this thesis is the language, CALL, through which computations and patterns are defined. We then discuss CALL's features and the advantages of a domain-specific language instead of using a standard programming language (including a database query language) in Section 9.2.

Section 9.3 we discuss whether this work is applicable in industry and the potential limitations of that application. Finally, in Section 9.4, we discuss possible evolutions of this work.

9.1 Limitations

The proposed framework alongside its declarative language, CALL, and supporting runtime, whose architecture was described in Chapter 3, were designed to support the purposes described in Chapter 1. As expected from a DSL and supporting runtime, they simplify the tasks they were designed to, but have limitations that are particularly noticeable in areas outside of the framework's intended scope.

We split the limitations of framework into three categories: theoretical, engineering and applicability.

Theoretical limitations refer to failures that cannot be identified by this framework or that cannot be correctly localized. These limitations are a hard bound on what can be performed with this framework without any fundamental changes to the approach. These are discussed in Section 9.1.1.

Engineering limitations are implementation limitations that could be overcome with a larger implementation effort. These are presented in Section 9.3 as they represent a cost to adoptability, but do not represent a fundamental limitation of the approach: rather, they are a limitation of the current implementation.

Applicability limitations arise from the framework having performance limitations making it unsuitable for some applications. For example, it is possible to use the framework and CALL to perform face recognition, although the cost and performance of such implementation would be abysmal. These limitations are a consequence of the mismatch between intended use of the framework and the problem to be solved. Section 9.1.2 discusses these limitations.

9.1.1 Theoretical Limitations

In its most abstract form, the proposed framework receives computations matched to elements in an architecture and outputs an ordered list of locations. The first limitation to what can be done with the framework is that it must match this structure. For example, an audio filtering system could not be implemented with the framework: even if the actual signal processing transformations could be represented in CALL, there is no way for the framework to output a signal as it only outputs probabilistic error localizations.

A more useful example of what *cannot* be done with the framework is anything that *acts* on a system. While the framework fits well within the "Analyze" phase of the MAPE loop, it cannot be used *in lieu* of the loop itself as it provides no mechanisms for monitoring, planning or execution.

Another limitation of the framework are patterns of computations that cannot be recognized and/or classified. CALL allows expression of logical statements, defined using predicate logic, over sets of computations. But, for example, probabilistic evaluation is not possible: CALL provides no way to write a predicate that matches computations with a certain probability rating stating, for example, that a computation a followed by a computation b represents a high-level computation c with probability 0.7 and high-level computation d with probability 0.3. These scenarios could be useful to reason with incomplete information when lack of probing requires a probabilistic matching instead of a strict matching, as imposed by CALL.

The language itself also has several limitions: (1) it supports only a limited number of data types, and (2) it has no primitives for I/O or OS integration. This means it is impossible to define CALL programs that read files or perform, for example, network access.

9.1.2 Applicability vs Expressiveness

While the theoretical limits of the framework and CALL discussed in Section 9.1.1 are very general, actual use of CALL is more constrained as several constructs are theoretically possible, but not usable in practice. In this section, we discuss the applicability and expressiveness trade-off. For a discussion on practicality, a parallel, but independent, concept from applicability, see Section 8.3.

Expressiveness is a measure of how straightforward it is to write a construct that can be described with CALL. Expressiveness is dependent on the feature to describe and the features of the language. For example, Java, without its standard library, does not have the concept of a

set, and does not have a concept of equality, so expressing a test to see if a structure belongs to a group is cumbersome:

```
public final class Check {
    public static boolean contains(int[] data, int value) {
        for (int i = 0; i < data.length; i++) {
          if (data[i] == value) {
                return true;
            }
        }
    }

    return false;
}
```

But, in CALL, this becomes simpler:

```
class check_t {
    static bool contains(set<int32> data, int32 value) {
        return value in data;
    }
}
```

In this scenario, checking whether a value is part of a group, CALL is more expressive than Java.

Expressiveness refers to how the features allow some concepts to be described. This is what drives common statements such as "perl is better at text file processing than Java". It is not that Java can't do it – it can –, but it is more cumbersome for the software developer.

Applicability refers to the degree to which declarations can be used in the real world. For example, it is possible to compute a cryptographic SHA digest of a file using CALL, by using lists of bytes to represent the digest and updating the digest from a list of bytes representing the file. However, the difficulty of implementing the constructs in CALL due to lack of expressiveness means it is not doable in practice.

Applicability and expressiveness are related: more expressiveness leads to more applicability, but it comes with other costs in practicality and complexity of implementation of the actual framework.

This means there are limitations in applicability of CALL that arise from some constructions being too cumbersome or error-prone to write, thereby making them hard to use. These limitations arise from expressiveness limitations in CALL. Having been designed to support the framework proposed, CALL makes it easy to write constructions that are common when relating behavior observations; other uses may require larger code bases or less intuitive designs. Unlike theoretical limits, the limits for applicability that arise from limitations in CALL's expressiveness are not hard bounds.

For example, in a system with 5 computation types all with a property `ID`, recognizing all 4-tuples of computations with the same `ID` requires a significant amount of engineering. Because recognizers are strictly typed, we would need $5^4 = 625$ different types of recognizers. While some constructions, like this one, require too much code to be usable, others are too complex. For example, implementing a cryptographic algorithm, such as SHA-1, would require very complex recognizers due to the lack of bitwise and shift operations in CALL. And others, while small and simple, are too inefficient in execution to be used. For example, using a stream as a dictionary and performing a lookup from values from another stream will be slow due to CALL's lack of

indexing and hashing.

Unfortunately, CALL also has limitations in expressiveness that make some tasks complex. Some of these are addressed in Section 9.4. For example, CALL does not allow recognizing sets with an arbitrary number of computations. Recognizers can only match a fixed number of computations. So, as in the example shown in Chapter 4 where we want to match a CCR-I, several CCR-Us and a CCR-T into a CDR, we are forced to use intermediate computation types and recognizers whose task is to aggregate these computations.

CALL is also missing several common functions to manipulate its data types, but those are engineering limitations (see Section 9.3). CALL also lacks a way of using existing libraries in other languages. This makes some checks difficult to do. What if we want to verify that a captcha returned by a web server is a valid jpeg image? There are no image manipulation functions in CALL and there is no way to use Java's or C++'s (see Section 9.3).

9.2 CALL *vs* Java and other alternatives

Given these limitations discussed in Section 9.1.2, one may question why introduce a new language and not just implement CALL as a framework that could be used from a mainstream programming language such as Java or C++?

For example, why declare a recognizer as:

```
recognizer foobar(f : foo, b : bar) {
    invariant { f.id == b.id }
    emit new foobar(f.id);
}
```

And not as:

```
class FoobarRecognizer extends Recognizer<Foobar, Foo, Bar> {
    @Override
    public Foobar matchAndEmit(Foo f, Bar b) {
        if (f.getId() == b.getId()) {
            return new Foobar(f.getId());
        }

        return null;
    }
}
```

The quick answer is that a framework in Java or C++ is possible, but undesirable as it makes syntax more complex and analysis more difficult. The extra burden on the syntax makes its use by domains experts more difficult and the analysis limitations makes the framework error-prone. If it removes some of the applicability limitations discussed in Section 9.1.2, it introduces others as will be seen in this section.

The difficulty with using Java or C++ for the framework language is a result of concept mismatch between the framework and mainstream languages. While *possible*, the syntax required when using mainstream languages is often more verbose and some concepts are difficult to match. Some guarantees are difficult to obtain and analysis is often more difficult if not impossible. For example, CALL ensures all invariants in recognizers always terminate. This is not possible to ensure in practice in Java without artificially constraining the language.

But not everything is difficult to map. Conditions, for example, exist in both languages:

```
// CALL
exists i : S @ i.id == test_id
```

```
// Java
return S.stream().hasAny(i -> i.id() == test_id)
// or, for a pre-Java 8 syntax:
for (Type s : S) {
    if (s.id() == test_id) {
        return true;
    }
}

return false;
```

```
// C++
return std::find(S.begin(), S.end(),
    [=](const Type &s) { return s.id() == test_id; }
```

As illustrated, CALL is consistently more compact. Its use of predicate logic makes expressing conditions and relationships easier: CALL is declarative – and hence you don't need an algorithm to be specified – you can leave that to the CALL compiler.

But while the gains for some concepts are small, for others the story is different. Neither Java nor C++ provide ways to implement invariants. To implement an equivalent of invariants in Java, we would have to write a significant amount of tedious and error-prone code:

So, instead of:

```
computation type comp_t {
    invariant x_non_negative { x >= 0 }
    int32 x;
}
```

We would have to write:

```
class Comp {
    private int x;
    public int getX() { return x; }
    public void setX(int x) {
        if (x < 0) {
            throw new IllegalArgumentException("x_non_negative");
        }

        this.x = x;
    }
}
```

Not using CALL as a language would also make it difficult to ensure parallelism of computations, a necessary conditions for scalability. Neither Java nor C++ have a built-in mechanism to ensure that computations can be executed in parallel without side-effects.

Again, we are discussing what can be done while maintaining applicability, as previously described. It is possible to implement anything in Java or C++ (CALL itself is implemented in Java). But the complexity required makes it impractical.

A different alternative to Java/C++ is using processing systems such as map/reduce. Map/reduce system, such as Apache Hadoop [13], provide a scalable way to process large numbers of computations. But many of these systems (and map/reduce is a good example), there is a trade-off between scalability, latency and expressiveness. While map/reduce is massively scalable, it

suffers from two limitations: (1) it has too much latency for use in our use case, and (2) it is more restricted in the computations it can perform. The increased latency is due to map/reduce requiring a large number of elements in the map operations for it to be efficient and this means collecting data for some significant amount of time before processing it. Map/reduce is used by high-data companies such as Google to perform offline processing, not online processing. The restrictions on the types of computations arise from the structure of map/reduce. Map/reduce performs computations that fit in a two/step paradigm: mapping data to a key and then joining (reducing) all values that were mapped to the same key. Only simple recognizers could be implemented in this way.

Another, perhaps better, alternative are Complex Event Processing systems. These are similar to CALL as they receive events use them to generate new events. If CALL had to be labeled with a "standard" system type, it would probably be labeled as a Complex Event Processing system. But, while Complex Event Processing systems provide fast and scalable matching, they do not provide several features that CALL does. Specifically, Complex Event Processing systems require fixed time windows to analyze events, are difficult for domain experts to understand and do not provide the analysis that CALL is capable of. Complex Event Processing systems are discussed in more detail in the Related Work chapter, in Section 7.4.

9.3 Adoptability in Industry

For the proposed framework to be adoptable by industry it needs to fulfill a few requirements: (1) it needs to solve a problem that industry has; (2) it must solve that problem in a way compatible with the industry's constraints; and (3) has to be usable by people in industry (this last requirement is an important specialization of the previous one).

(1) and (3) are directly addressed throughout this thesis (see Chapters 1 and 2 and Section 8.3) and should not need further discussion.

The Samsung case study shows that (2) is, at least in broad terms, fulfilled. CALL was connected to an event bus that reproduced the real one in Samsung, was limited to probing constraints Samsung had in its production systems, handled the amount of traffic required by Samsung and used the correctness criteria provided by Samsung. However, there are some limitations of CALL that could be overcome in a straightforward way:

Providing Common Infrastructure CALL provides ways to reuse code between implementations, but, for industry adoption, an actual repository of code to reuse needs to be written with CALL libraries and pre-built probes.

We promise (and deliver) that CALL is modular and provides easy reuse of common patterns of events. We demonstrate how to write them in CALL and use these patterns in some scenarios in this thesis (see Chapter 4). But the current CALL implementation provides libraries for only some common styles, such as call/return, but not for others, such as repository [22, 71].

Also, implementation of common protocols in CALL is currently limited to a subset of HTTP. Other forms of IPC (shared memory, JSON/HTTP, SOAP) are not provided.

Industry adoption of CALL *would require at least some more libraries to be written to provide good coverage of commonly used styles and protocols.*

We argue that probes are system-specific and that CALL provides an architecture/OS/language agnostic way to connect them to the framework. This allows probes to run in multiple hardware nodes, in different operating systems and to be implemented in multiple languages. But probes have to be developed for every system and some common ones would make CALL more attractive. For example, probes that read logs, HTTP interceptors, J2EE filters are examples of simple very general probes that would lower the cost of adoption.

Industry adoption of C*ALL would require some standard probes to be provided with* C*ALL.*

Expressiveness Increased expressiveness of CALL would make industry adoption easier. Currently, CALL has support only for types and no support for "external types" (data types whose values are not handled by CALL, but by external code). CALL's runtime was designed to support these and CALL's compiler and runtime support these.

Linking in CALL libraries that support common functions such as HTTP URL encoding/decoding, JSON and XML parsing/extraction, image parsing or file access would increase significantly the range of applications CALL could be used for.

Industry adoption of C*ALL would require some common external types to be implemented.*

CALL allows reasoning about tuples of computations in the context of streams. However, in practice, several instances arise in which reasoning about sets of computations is needed. This topic is discussed in more detail in Section 9.4.

Performance Recognizers keep computations in their queues until they can be discarded. When new computations "arrive" from probes or other recognizers, they are matched against all existing computations. CALL lacks any type of indexing in the recognizer inputs. Recognizer performance will degrade exponentially with the number of computations not yet discarded.

There are no known ways to reduce the set of elements to match: relational databases and complex event processing systems all suffer from this performance degradation when queries are applied to large data sets. However, what relational database engines and complex event processing engines do that CALL does not is indexing the elements to match so that search is done more efficiently. In a comparison with relational database engines, CALL does the equivalent of full table scans for every query.

There are, however, no theoretical reasons why the same optimizations performed in relational databases and complex event processing systems could not be applied to CALL. While CALL's structure is based on predicate logic and not relational calculus as relational databases, the equivalence between both is well established.

Industry adoption of C*ALL would require indexing of recognizer buffers to be implemented.*

Similar to indexing, recognizers do not support any intelligent use of the data. If indexing is added, then use of indexes requires query planning capabilities. Just like indexing, this is not a theoretical issue, but definitely a practical one.

Industry adoption of C*ALL would require query planning in recognizers be implemented.*

9.4 Future Work

This thesis proposes a framework identification and localization of failures at run-time, based on a model of system behavior. It relies on several assumptions that can be challenged and analyzed in future work.

Some future work has already been outlined in Section 9.3. This section will focus on future work that is not straightforward engineering.

9.4.1 Augmenting CALL Expressiveness

Future work can be done to improve the framework's practicality and capabilities. As seen in the example in Section 4.2, CALL is easy to use to define recognizers that recognize a high-level computation from a fixed number of low-level computations. This has been successfully used in both case studies. However, as seen also in Section 4.2, recognizing a high-level computation from an *arbitrary number* of low-level computations requires defining intermediate computation types and recognizers.

Future work can also be done to make CALL easier to use, reducing the language. CALL has several constructs that are specific to the problem domain (*e.g.*, recognizers, oracles, matching invariants). But CALL also has several constructs that are not domain-specific like the syntax to create methods in classes. Better integration with programming languages would leverage better tools and simplify CALL to the core it excels in.

9.4.2 MAPE Loop Integration

The further research paths described so far challenge some elementary mechanisms in the proposed framework. But work can also be done in integrating the framework with higher-level systems that are handling the MAPE loop.

Specifically, no work was done to show how such a framework would integrate with a MAPE-loop adaptation framework. It was said, in the introduction, that this work fits in the analysis part of the loop, but not how it would be used by the other parts of the loop. In particular, how would the results of this framework change planning and execution? And vice versa? This is a particularly rich area, where multiple evolutions can be seen.

Could uncertainty on the health status of components drive dynamic testing? If some component has not been exercised in some time, could we forcibly exercise it to ensure it is working and take corrective action if it isn't? This would be an approach similar to Netflix's Chaos Monkey [57].

9.4.3 Increasing Autonomy

The proposed framework relies on humans to define recognizers and, therefore, explicitly identify behavior patterns. But machine learning research and practice has proven that many patterns can be inferred (or "learned") from observing behavior. Leveraging machine learning to define recognizers based on observed patterns is a possible path for future work. One immediate difficulty with such an approach would be handling intermediate abstractions. Recognizing

high-level computations from low-level observations in a single step may yield very complex recognizers and generate pattern-matching rules that are hard to understand, making such systems hard to debug. But having machine learning to identify computations with increasing levels of abstraction is also challenging.

A similar path for future work is related to oracles. Just like for recognizers, we could leverage machine learning to infer which computations are correct and incorrect. Supervised machine learning seems to be a promissing path for this, but it would need further research to understand if supervised learning could actually be simpler and/or less cumbersome than manual definition of oracles.

We could also extend the notion of correctness to include fuzziness or certainty. This may be a very useful extension if we introduce machine learning, but may also be useful without it. Current oracles are binary: computations are either correct or incorrect. This limits the type of information that can be presented to the MAPE loop. A failing server may decrease total system response time, but the degradation may not reach a high-enough threshold to require corrective action.

9.4.4 Sharding Support

Sharding is a form of parallelism that is different from the one currently supported by CALL-RE. When sharding, we have a whole copy of the system dealing with a subset of the computations and another one dealing with another subset. Sharding is more scalable than the model proposed in this thesis because it allows almost unlimited parallelism.

Sharding introduces two difficulties: splitting the computations into groups that are independent of each other and then joining the results.

In some limited scenarios both these problems may be solved by the structure of the system itself. A web serving system that has two independent physical locations can monitor each location independently. In these cases joining the results is also trivial as the components of the system are also independent.

But if the original system structure isn't itself sharded, identifying how can a system be split can be more difficult. CALL has rules to combine computations so it may be possible to determine how computations can be sharded by looking at recognizer rules.

9.4.5 Dynamic probe placement

Throughout this thesis, all examples we've seen have used a fixed diagnosis system. The probes, recognizers and oracles are all fixed and the runtime architecture was fixed. But CALL allows the architecture to change and allows the actual probes deployed to be changed. This means the MAPE loop can not only adapt the target system, but it could adapt its own diagnosis.

By analyzing traces using the work in [19], it is possible for the MAPE loop to realize which components can be individually diagnosed and not. This allows running a system with reduced probing costs as long as there are no failures, and deploy additional probes when a failure is detected. Depending on the complexity of the target system, it is possible to deploy probes in several phases to pinpoint the source of failure iteratively.

For certain systems with high tolerance to failures, or low likelihood of non-intermittent faults, probes could be disabled completely and regularly deployed to test parts of the system. A grid computing farm could use such a technique to regularly evalute the quality or performance of its nodes without paying the cost of regular monitoring.

All these approaches would need to be evaluted in the context of a MAPE loop. It would require research on how to balance the cost of probing, cost of deploying the probes, the time taken to deploy and diagnose failures and the cost of failure itself.

Chapter 10

Bibliography

[1] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In Vadim Bulitko and J. Christopher Beck, editors, *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, Lake Arrowhead, California, USA, 8 – 10 July 2009. AAAI Press. 7.2

[2] Rui Abreu and Arjan J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010. 7.2, 8.2

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: http://dx.doi.org/10.1109/ASE.2009.25. URL `http://dx.doi.org/10.1109/ASE.2009.25`. 3.5, 3.5, 3.7, 7.2, 8.2

[4] Y. Al-Nashif, A.A. Kumar, S. Hariri, Guangzhi Qu, Yi Luo, and F. Szidarovsky. Multi-level intrusion detection system (ml-ids). In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 131–140, 2008. doi: 10.1109/ICAC.2008.25. 7.1

[5] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144. 7.3

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126 (2):183–235, April 1994. ISSN 0304-3975. doi: 10.1016/0304-3975(94)90010-8. URL `http://dx.doi.org/10.1016/0304-3975(94)90010-8`. 7.3

[7] Siddharth Anand. Keeping movies running amid thunderstorms. `http://www.infoq.com/presentations/Keeping-Movies-Running-Amid-Thunderstorms`, 2011. 1

[8] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004. A

[9] Mario R. Barbacci, Robert J. Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. Quality attribute workshops qaws - third edition. Tech-

nical Report CMU/SEI-2003-TR-016, Carnegie Mellon, 2003. URL http://www.sei.cmu.edu/library/abstracts/reports/03tr016.cfm. 3.1

[10] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *World Congress on Formal Methods*, 2012. 7.3

[11] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62:1 – 45, 2015. 7.3

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, second edition*. SEI Series in Software Eng. Addison-Wesley Professional, 2003. 3.1, 6.4

[13] Dhruba Borthakur. Hdfs architecture guide. http://hadoop.apache.org/common/docs/current/hdfs_design.html, 2012. 9.2

[14] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984. ISSN 0004-5411. doi: 10.1145/828.833. URL http://doi.acm.org/10.1145/828.833. 7.3

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proc. OSDI'04*, San Francisco, CA, 2004. 7.2

[16] Paulo Casanova, Bradley R. Schmerl, David Garlan, and Rui Abreu. Architecture-based run-time fault diagnosis. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2011. ISBN 978-3-642-23797-3. 1, 8.1.1, 8.1.2, 8.1.2, 8.1.2

[17] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. Diagnosing architectural run-time failures. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 20-21 May 2013. To appear. 1

[18] Paulo Casanova, Bradley Schmerl, David Garlan, Rui Abreu, and Jungsik Ahn. Applying autonomic diagnosis at samsung electronics. Technical Report CMU-ISR-13-111, Institute for Software Research, Carnegie Mellon University, 2013. 1

[19] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. Diagnosing unobserved components in self-adaptive systems. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 75–84, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2864-7. doi: 10.1145/2593929.2593946. URL http://doi.acm.org/10.1145/2593929.2593946. 1, 3.5, 3.6.2, 8.1.3, 8.2, 8.2, 9.4.5

[20] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, chapter Software Engineering for Self-Adaptive Systems: A Research

Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02160-2. doi: 10.1007/978-3-642-02161-9_1. URL http://dx.doi.org/10.1007/978-3-642-02161-9_1. 1, 1

[21] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of SEAMS'06*, 21-22 May 2006. 7.2

[22] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures, second edition*. Addison-Wesley, 2010. 3.1, 6.4, 9.3

[23] C++ Committee. C++ standard template library. http://www.cplusplus.com/reference/stl/. **??**

[24] Boost Community. Boost library. http://www.boost.org/,. **??**

[25] Pygmy Community. Pygmy webserver. https://pygmy-httpd.sourceforge.net/,. 8.1.1

[26] Doug Cutting. The hadoop framework. http://hadoop.apache.org/, 2010. 7.2

[27] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. ISSN 0004-3702. 7.2

[28] Rogerio de Lemos, Holger Giese, Hausi Müller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cikic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Poala Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Wilhelm Schlichting, Bradley Schmerl, Dennis B. Smith, Joao P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, number 10431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2011/3156. 1, 1

[29] Nicholas Digiuseppe and James A. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Softw. Engg.*, 20(4):928–967, aug 2015. ISSN 1382-3256. doi: 10.1007/s10664-014-9304-1. URL https://doi.org/10.1007/s10664-014-9304-1. 7.2

[30] EsperTech. Esper – complex event processing. http://www.espertech.com/. 7.4

[31] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language (aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006. 7.3

[32] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, November 1997. 4.1.1, 7.3

[33] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004. 3.1, 3.4, 7.2

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Symposium on Operating Systems Principles*, New York, NY, USA, 2003. ACM. 1, 3.1, 7.2

[35] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, January 2007. ISSN 0167-9236. doi: 10.1016/j.dss.2006.06.011. URL `http://dx.doi.org/10.1016/j.dss.2006.06.011`. 7.1

[36] Google. Google guava library. `https://github.com/google/guava`, 2018. **??**

[37] O. M. G. Group. UML Specification, Version 2.0. 7.3

[38] Jin Heo and Tarek Abdelzaher. Adaptguard: Guarding adaptive systems from instability. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 77–86, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: 10.1145/1555228.1555256. URL `http://doi.acm.org/10.1145/1555228.1555256`. 7.2

[39] IBM. An architectural blueprint for autonomic computing, fourth edition, 2006. 1

[40] ISO/IEC 13568. Information technology – Z formal specification notation – Syntax, type system and semantics, 2002. 7.3

[41] ISO/IEC 14882:2011. Information technology – Programming languages – C++, 2011. 7.3

[42] 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, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL `http://doi.acm.org/10.1145/1101908.1101949`. 3.5, 8.2

[43] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE'02*. ACM Press, May 2002. 7.2

[44] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125, 2017. doi: 10.1109/QRS.2017.22. 7.2

[45] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1): 41–50, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055. 1

[46] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. of FTCS'95*, Washington, DC, USA, 1995. IEEE Computer Society. 7.2

[47] Jeff Kramer and Jeff Magee. A rigorous architectural approach to adaptive software engineering. *J. Comput. Sci. Technol.*, 24:183–188, March 2009. ISSN 1000-9000. doi: 10.1007/s11390-009-9216-5. URL `http://portal.acm.org/citation.cfm?`

id=1599001.1599003. 7.2

[48] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 7.3

[49] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. of PLDI'05*, Chicago, Illinois, USA, 2005. 7.2

[50] C. Liu, L. Fei, X. Yan, J. Han, and S.P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering (TSE)*, 32(10):831–848, 2006. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2006.105. 7.2

[51] Doug Madory. Facebook's historic outage, explained. https://web.archive.org/web/20220104172609/https://www.kentik.com/blog/facebooks-historic-outage-explained. 1

[52] Alessandro Margara and Gianpaolo Cugola. Processing flows of information: from data stream to complex event processing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 359–360, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0423-8. doi: 10.1145/2002259.2002307. URL http://doi.acm.org/10.1145/2002259.2002307. 7.4

[53] Mathworks. Simulink. http://www.mathworks.com/products/simulink. 7.3

[54] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *Proc. of ASE'08*, 2008. 7.2

[55] Patrick O'Neil Meredith. *Efficient, expressive, and effective runtime verification*. University of Illinois at Urbana-Champaign, 2012. 7.3

[56] Rich Miller. Failure rates in google data centers. http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers. 1

[57] Netflix. Chaos monkey. https://netflix.github.io/chaosmonkey. 9.4.2

[58] OWASP. Owasp top 10. https://owasp.org/www-project-top-ten, 2021. 1

[59] Marko Palviainen, Antti Evesti, and Eila Ovaska. The reliability estimation, prediction and measuring of component-based software. *Journal of Systems and Software*, 84(6):1054 – 1070, 2011. ISSN 0164-1212. doi: DOI:10.1016/j.jss.2011.01.048. URL http://www.sciencedirect.com/science/article/pii/S0164121211000380. 7.2

[60] Soila Pertet and Priya Narasimhan. Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, page 48, 2005. 3.1

[61] David M. W. Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation. Technical Report SIE-07-001, School of Informatics and Engineering, Flinders University, Adelaide, Australia, 2007. 8.1.3

[62] Akshay Rajhans, Ajinkya Y. Bhave, Sarah Loos, Bruce Krogh, Andre Platzer, and David

Garlan. Using parameters in architectural views to support heterogeneous design and verification. In *50th IEEE Conference on Decision and Control (CDC) and European Control Conference (ECC)*, Orlando, FL, 12-15 December 2011. 7.3

[63] Giles Reger and Helena Cuenca Cruz. Monitoring at runtime with qea. 2015. 7.3

[64] Dave Rensin and Adrian Hilton. How to avoid a self-inflicted ddos attack - cre life lessons. https://cloud.google.com/blog/products/gcp/how-to-avoid-a-self-inflicted-ddos-attack-cre-life-lessons. 1

[65] Jesus Rios, Saurabh Jha, and Laura Shwartz. Localizing and explaining faults in microservices using distributed tracing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 489–499, 2022. doi: 10.1109/CLOUD55607.2022.00072. 7.1, 7.2

[66] Paul Robertson and Robert Laddaga. Model based diagnosis and contexts in self adaptive software. In Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad Moorsel, and Maarten Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26009-7. doi: 10.1007/11428589_8. URL http://dx.doi.org/10.1007/11428589_8. 7.1

[67] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009. ISSN 1556-4665. doi: 10.1145/1516533.1516538. URL http://doi.acm.org/10.1145/1516533.1516538. 7.1

[68] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022. doi: 10.1109/ACCESS.2022.3144079. 7.2

[69] Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *Winter Simulation Conference*, WSC '09, pages 216–229. Winter Simulation Conference, 2009. ISBN 978-1-4244-5771-7. URL http://dl.acm.org/citation.cfm?id=1995456.1995492. 7.3

[70] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. Univ of Illinois Press, 1949. ISBN 0-252-72548-4. 3.5

[71] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996. 9.3

[72] Amazon AWS Team. Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648,. 1

[73] Amazon AWS Team. Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region. https://aws.amazon.com/message/41926,. 1

[74] Kishor S. Trivedi and Kalyanaraman Vaidyanathan. Software aging and rejuvenation. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008. 7.2

[75] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based

fault localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 365–376, 2021. doi: 10.1109/SANER50967.2021.00041. 3.5, 7.2

[76] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070536. URL http://dx.doi.org/10.1109/ICSE.2009.5070536. 7.2

[77] Wikipedia. Google services outages. https://web.archive.org/web/20230314194507/https://en.wikipedia.org/wiki/Google_services_outages, 2023. 1

[78] W. ERIC WONG and YU QI. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009. doi: 10.1142/S021819400900426X. URL http://www.worldscientific.com/doi/abs/10.1142/S021819400900426X. 3.5

[79] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004. 3.4, 8.1.1

[80] Qixun Zhang, Tong Jia, Zhonghai Wu, Qingxin Wu, Lichun Jia, Donglei Li, Yuqing Tao, and Yutong Xiao. Fault localization for microservice applications with system logs and monitoring metrics. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 149–154, 2022. doi: 10.1109/ICCCBDA55098.2022.9778893. 7.1, 7.2

[81] Teng Zhang, Ramneet Kaur, Insup Lee, and Oleg Sokolsky. Runtime verification of parametric properties using smedl. In *From Reactive Systems to Cyber-Physical Systems*, 2019. 7.3

Appendices

Appendix A

Terminology: faults, errors and failures

This section defines basic terminology that is commonly used in dependable computing. We will commonly refer to faults, errors and failures and it is useful to provide some clarification on the meaning of these terms, which derive from [8].

Faults are the original source of incorrect behavior. They correspond, intuitively, to incorrect implementation of a system. For example, software bugs and hardware problems are typical examples of faults.

Faults can only affect the system when exercised. Exercising a software bug means executing the buggy code and exercising a hardware failure means running the faulty hardware. Faults that do not affect the system even when they are exercised are known as *dormant*. Dormant faults cannot be detected by observation of system behavior. A fault becomes *active* if, when exercised, produces an *error*. An *error* is a violation of some condition in, or assumption about, the system's state. Errors are not directly visible in general, as the system's state is not commonly directly monitored. However, this corrupted state may, in turn, lead to an observable violation of the system's specification yielding a *failure*.

The distinction between these concepts is important: exercising the fault and detecting the failure may or may not be close in time; an active fault may generate an error that manifests itself only in a later computation. For example, consider a bug in the web server that erroneously deletes a required image from the web server's file system: when this fault is exercised, it will corrupt the file system. It is only later, when the image is actually requested by a web client, that a failure occurs as the web server performs incorrect behavior by returning an error code.

Faults that switch between being *dormant* and *active* are known as *intermittent faults*. For example, an algorithm that checks whether a number is strictly positive using the formula $x \geqslant 0$ will generate an error only when $x = 0$. If the code is executed with $x = 1$ the fault is *dormant*, as it will not affect system behavior. But if the code is executed with $x = 0$ then the fault will affect system behavior. However, if the formula were $x \leqslant 0$ then the fault would not be intermittent, as it would always produce the wrong value.

An important special case of intermittent faults is *correlated faults*. A set of correlated faults will manifest itself only when *all* faults become active simultaneously. Suppose one of the web servers in figure 2.1 has a power failure and is incapable of processing requests. In a correct system, the dispatchers will not send requests to the faulty web server, so this fault will not generate failures. If a dispatcher has a bug and, as a consequence of exercising the fault, forwards

a request to the stopped web server, these two faults, *combined* will yield a failure.

Appendix B

CALL Syntax

This chapter describes, informally, the syntax of CALL. The syntax is described bottom up, starting with the language's more elementary concepts and moving to the higher concepts. We start by describing data types in Section B.1, followed by expressions in Section B.2 and then operations in Section B.3.

B.1 Data types

CALL supports data types that can be divided into the following categories:

- Primitive types (Section B.1.1);
- Enumeration types (Section B.1.2);
- Complex types (Section B.1.3);
- Structure types (Section B.1.4);
- Operable types (Section B.1.5).

Data types in CALL are all hierarchical in the object-oriented sense: inheritance means substitutability.

B.1.1 Primitive data types

CALL defines the following primitive data types:

- `any`. The `any` data type is an abstract data type and is the supertype for all types defined in CALL.

```
1 any variable_name;
```

Listing B.1: Declaration of a variable of `any` data type.

- `ascii`. The `ascii` data type contains strings with only ASCII characters. From a purely computation perspective, there is no advantage in using `ascii` instead of `string`. However, for interfaces with external systems, namely with sensors reporting events, `acsii` may be much simplier to use.

- `string`. The `string` data type contains strings containing any character that can be encoded in UTF-8.

```
1 ascii ascii_var = "some text";
2 string string_var = u"some non-ascii text";
```

Listing B.2: Declaration with assignment of a variable of `ascii` and `string` data types.

- `bool`. The `bool` data type represnts a boolean value. It can have values `true` or `false`.

```
1 bool bool_true_var = true;
2 bool bool_false_var = false;
```

Listing B.3: Declaration with assignment of variables of `bool` data type.

- `bytearray`. The `bytearray` data type represents a list of bytes. CALL does not provide many operations on `bytearray`. However, it is useful to allow general binary data to be reported by sensors and processed by custom event handlers. Additionally, custom functions may be defined to process binary data.

```
1 bytearray bytes_var;
```

Listing B.4: Declaration of variables of `bytearray` data type.

- `float`. The `float` data type contains a single-precision floating point number. It has the same semantics as in the Java language.
- `double`. The `double` data type contains a double-precision floating point number. It has the same semantics as in the Java language.

```
1 float floats_var = 5.6f;
2 double double_var = 5.6;
```

Listing B.5: Declaration and assignment of variables of `float` and `double` data types.

- `int8`. An `int8` represents an integer number with 8 bits with values ranging from -2^7 to $2^7 - 1$.
- `int16`. An `int16` represents an integer number with 16 bits with values ranging from -2^{15} to $2^{15} - 1$.
- `int32`. An `int32` represents an integer number with 32 bits with values ranging from -2^{31} to $2^{31} - 1$.
- `int64`. An `int64` represents an integer number with 64 bits with values ranging from -2^{63} to $2^{63} - 1$.

```
1 int8 var8 = 5b;
2 int16 var16 = -6h;
3 int32 var32 = 7;
4 int64 var64 = -8l;
```

Listing B.6: Declaration of variables of `int8`, `int16`, `int32` and `int64` data types.

- `time`. The `time` data type contains a timestamp measured in microseconds.
- `period`. The `period` data type contains a time span measured in microseconds.

```
time time_instant;
period one_second = 1s;
period one_millisecond = 1ms;
period one_microsecond = 1us;
```

Listing B.7: Declaration of variables of `time` and `period` data type.

B.1.2 Enumeration data types

Enumerations provide predefined lists of values.

```
enum happiness {
    crazy;
    very_happy;
    sad;
    writing_thesis;
}

//
happiness my_happiness = writing_thesis;
```

Listing B.8: Declaration of enumeration data types and declaration and assignment of enumeration variables.

B.1.3 Complex data types

CALL defines the following complex data types:

- **sets**. A `set` is an unordered collection of items where duplicate items are not allowed. Sets are represented with a special notation to show what type of data type they contain. All sets are subtypes of `any`.
- **bags**. A `bag` is an unordered collection of items where duplicate items are allowed. Bags are represented with a special notation to state what is the data type of the values they contain. All bags are subtypes of `any`.
- **lists**. A `list` is an ordered collection of items. Lists are represented with a special notation to state what is the data type of the values the list contains. All lists are subtypes of `any`.

```
set<int32> integer_set = {4, 5, 6};
bag<int32> integer_bag;
list<int32> integer_list = [1, 2, 3];
```

Listing B.9: Declaration and assignment of variables of `set`, `bag` and `list` data types.

- **maps**. A `map` is a partial function that, for some input values produces output values. Maps are represented with a special notation stating what types are used as keys (domain values) and values. All maps are subtypes of `any`.
- **tuples**. A `tuple` data type is a type containing an ordered list of types. Each instance of a tuple contains one instance of each value in the tuple. Tuples are subtypes of `any`.

```
map<int32, string> maps_integer_to_string;
tuple<int32,string,bool> tuple_of_3_types;
```

Listing B.10: Declaration of variables of `map` and `tuple` data types.

- **optionals**. An optional data type whose instances optionally contain values of another data type. Optional data types are marked with a special notation and are subtypes of `any`.

```
int32? optional_int_with_value = 14;
int32? optional_int_without_value = null;
int32?? optional_optional_int;
```

Listing B.11: Declaration and assignment of variables with optional values.

B.1.4 Structure data types

Structures define complex data types where one or more data types are associated with a name. For example, the following listing contains a declaration of a structure data type named `point` that contains two fields: an integer named `x` and an integer named `y`.

```
struct point {
    int32 x;
    int32 y;
}
```

Listing B.12: Declaration of structure `point`.

By default, structures are subtypes of `any`. However, structures can form hierarchies and, in that case, a structure will have the structures it inherits from as supertypes.

```
struct time_interval {
    time start;
    time end;
}

struct point_in_time : point, time_interval {
}
```

Listing B.13: Declaration of structure hierarchies.

The structure `point_in_time` will contain the four inherited fields and will be a subtype both of `point` and `time_interval`.

Structures can be declared abstract, meaning values of those types cannot be created, although values of subtypes of those structure may be created.

```
abstract struct abstract_structure {
    int32 value;
}
```

Listing B.14: Declaration of structure hierarchies.

B.1.5 Operable data types

Operable data types define data types that can have operations. There are two types of operable data types: *typedefs* and *classes*.

A typedef defines a restriction of another data type. Typedefs may include operations (the syntax for operations is described in Section B.3) and invariants.

An invariant defines a condition that must always be true in a data value. More information on invariants is provided in Section B.3.2.

The following listing contains an example of a `typedef` defining a restriction on integers:

```
1 typedef positive32 restricted int32 {
2     invariant negative_not_allowed { this >= 0 };
3
4     expr bool is_zero() {
5         return this == 0;
6     }
7 }
```

Listing B.15: Example of a typedef.

This code defines a new data type named `positive32`. `positive32` is a new data type, it is *not* an `int32`, although implicit conversion will happen in some scenarios as described in Section B.2.8.

This data type is a restriction of the `int32` data type. Only `int32` data values that validate the provided invariant will be allowed in the data type. Multiple invariants may be defined and they must all be validated.

In addition to the invariant, the `positive32` data type defines an *operation*, named `is_zero`. The syntax for operations is described in Section B.3.

Classes define data types that are similar to structures but, like typedefs, can contain invariants and operations. The following listing contains an example of a class declaration:

```
1  class point_with_altitude : point {
2      int32 altitude;
3
4      invariant there_are_limits_for_holes { altitude >= -5000000 };
5
6      expr bool is_above_ground() {
7          return alitude >= 0;
8      }
9
10     expr bool is_same_height(point_with_altitude p) {
11         return altitude == p.altitude;
12     }
13 }
```

Listing B.16: Example of a class.

Classes can have other classes or structures as supertypes. However, restricting a supertype's value in an invariant is not allowed.[1] Classes can, instead of inheriting from other classes or structs, restrict other classes or structures. Consider the following example:

```
1 class timed_northern_hemisphere_point : time_interval, (point_with_altitude) {
2     invariant only_in_northern_hemisphere { x >= 0 }
3 }
```

Listing B.17: Example of a class with restrictions.

The `timed_northern_hemisphere_point` is a subtype of `time_interval` but not of `point_with_altitude`. However, it contains a "copy" of the `point_with_altitude`

[1]This is not actually enforced in the language.

class: it has an `altitude` field, a `there_are_limits_for_holes` invariant and a `is_above_ground` and `is_same_height` methods. This syntax, similar to inheritance, but with the class name in parenthesis, is called "copying".

Copying is required when the substitutability principle is not applicable. The `point_with_altitute` class is itself a `point`, a structure with fields `x` and `y` with no restrictions on their values. If `timed_northern_hemisphere_point` were a subclass of `point`, then it could be cast to `point` and some code could use it as if it were a `point`. This code could then set its coordinates to somewhere in the southern hemisphere, unexpectedly violating the class's invariant. By copying, instead of inheriting, we do not allow casting from `timed_northern_hemisphere_point` to `point`.

But we do allow conversion between the two classes (see Section B.2.8 for a discussion between casting and conversion). In fact, conversion from `timed_northern_hemisphere_point` to `point_with_altitude` is done automatically:

```
1  void do_something(timed\_northern\_hemisphere\_point p) {
2    point pt = p; // conversion is automatic.
3    pt.y = pt.x;  // no invariants to verify in point.
4
5    // This also works:
6    if (is_on_x_axis(p)) {
7    }
8  }
9
10 bool is_on_x_axis(point p) {
11   return p.y == 0;
12 }
```

Listing B.18: Example of a conversion with copying

B.1.6 Namespaces

Data types in CALL can be organized in namespaces, which are used as a way to avoid naming clashes.

```
1  struct xpto {
2      xpto myself;
3
4      group1::xpto xpto_in_group_1;
5      ::group1::xpto also_xpto_in_group_1;
6
7      group2::xpto xpto_in_group_2;
8      ::group2::xpto also_xpto_in_group_2;
9
10     group2::sub::xpto xpto_in_group_2_sub;
11     ::group2::sub::xpto also_xpto_in_group_2_sub;
12 }
13
14 namespace group1 {
15     struct xpto {
16         xpto myself;
17         ::group1::xpto also_myself;
18
19         ::xpto outside_xpto;
20
21         ::group2::xpto xpto_in_group_2;
22
23         ::group2::sub::xpto xpto_in_group_2_sub;
```

```
    }
}

namespace group2 {
    struct xpto {
        xpto myself;
        ::group2::xpto also_myself;

        ::xpto outside_xpto;

        ::group1::xpto xpto_in_group_1;

        sub::xpto xpto_in_group_2_sub;
        ::group2::sub::xpto also_xpto_in_group_2_sub;
    }

    namespace sub {
        struct xpto {
            xpto myself;
            ::group2::sub:: xpto also_myself;

            ::xpto outside_xpto;

            ::group1::xpto xpto_in_group_1;

            ::group2::xpto xpto_in_group_2;
        }
    }
}
```

Listing B.19: Example of namespaces.

B.2 Expressions

Expressions are language constructs that evaluate at runtime to a value. Expressions in CALL are always typed (including `null`), meaning, every expression always results in values that can be assigned to a specific data type.

Expressions in CALL are divided into several categories:

- Literal expressions. See Section B.2.1.
- Variable expressions. See Section B.2.2.
- Operation expressions. See Section B.2.3.
- Collection expressions. See Section B.2.4.
- Quantifier expressions. See Section B.2.5.
- Field access expressions. See Section B.2.6.
- Function and method invocation expressions. See Section B.2.7.
- Casting and conversion expressions. See Section B.2.8.

B.2.1 Literal Expressions

Literal expressions are expressions that evaluate to fixed values that can be computed at compile time. There are literal expressions corresponding to most data types in CALL.

The following listing contains examples of all literal expressions for primitive data types:

```
bool true_value = true;
bool false_value = false;

int8 byte_value = 4b;
int16 short_value = 5h;
int32 int_value = 6;
int64 long_value = 7l;

float float_value = 1.2f;
double double_value = 3.4;

period one_microsecond = 1us;
period one_millisecond = 1ms;
period one_second = 1s;

ascii ascii_text = "foo";
string utf8_text = u"foo";
```

Listing B.20: Examples of literal expressions for primitive data types.

The syntax for creating collection literals is more complex. The two simplest cases are sets and lists, shown in the listing below.

```
set<int32> a_set = { 12, 23 };
list<int32> a_list = [ 34, 45 ];
```

Listing B.21: Examples of literal expressions for sets and lists.

The literal expression is the right-hand side of the expressions. CALL's type inference is (unfortunately) not very sophisticated so the type of the set or list is deduced from the types of the items in it. The left-hand side of the assignment is not taken into consideration when determining the set or list type. However, a common type for all items in the set or list will be used (see Section B.2.10 for a discussion of how *unification*, the process of finding a common data type, is done). This means that, for example, the following expression is valid:

```
set<int32> a_set = { 1b, 2h, 3 };
```

Listing B.22: Examples of literal expressions for sets and lists.

Because `1b` is a `int8`, `2h` is a `int16` and `3` a `int32`, and because the type that results from unifying the three is `int32`, the resulting right-hand side is `set<int32>`.

However, the type of the right-hand side may be forced using the `|:` syntax. This is also empty collections can be created, since the contained data type for literal empty collections cannot be inferred.

```
set<int32> a_set = { 1b, 2b }|:set<int32>;
list<period> a_list = []|:list<period>;
```

Listing B.23: Examples of literal expressions for sets and lists.

The same syntax is used to create literal bags. Bags are declared as a set but with the specific bag data type:

```
bag<int32> a_bag = { 1, 1, 2, 2, 2, 3 }|:bag<int32>;
```

Listing B.24: Examples of literal expressions for sets and lists.

In CALL, `null` can be used as a literal value. Unless a specific type is used using `|:`, `null` assumes the type `any?`. The following examples demonstrate `null` literals:

```
1 any? default_null_type = null;
2 int32? optional_int = null |: int32;
```

Listing B.25: Examples of literal expressions for sets and lists.

B.2.2 Variable Expressions

Variables have been shown in several examples. Variable expressions are expressions that evaluate to a value. The value of variable expressions is determined at run-time, although its type is known at compilation time. Variable expressions can be used inside collection literal expressions.

```
1 int32 a = 4;
2 int32 b = a;
3 set<int32> c = { a, b, 5 };
```

Listing B.26: Examples of literal expressions for sets and lists.

B.2.3 Operation Expressions

Operation expressions are expressions that are compose other expressions. There are three types of operation expressions:

- Arithmetic Expressions. See Section B.2.3.
- Comparison Expressions. See Section B.2.3.
- Boolean Expressions. See Section B.2.3.

Arithmetic Expressions Arithmetic expressions are: add, subtract, multiply, divide and modulo. Table B.1 describes the operators and provides examples. Note that due to automatic data type conversion (see Section B.2.8), in practice many more combinations of data types can be used with these operators.

Comparison Expressions Comparison expressions are expressions that compare values and return a boolean output. There are several type of comparison expressions in CALL: equality (==), inequality (!=), greater (>), greater or equal (>=), lesser (<), lesser or equal (<=), is null (`is null`), is not null (`is not null`).

With exception of the `is null` and `is not null` operators, all other comparison operators can be applied to `int8`, `int16`, `int32`, `int64`, `float`, `double`, `time`, `period`, `ascii` and `string`. In all cases, the left and right-hand sides must have the same data type.

Due to automatic casting and conversion (see Section B.2.8), these operators may be used with different data types. For example, it is legal to write `3 != 4h`. However, this is not a comparison between an `int32` and an `int16`. In this expression, `4h` is first converted to `4` and the comparison occurs between `int32` data types.

Operator	LHS[1]	RHS[2]	Result	Notes
+	int[3]	int	int	LHS and RHS must have the same type; result has the same type as both sides.
+	float	float	float	
+	double	double	double	
+	period	period	period	
+	time	period	time	
−	int	int	int	LHS and RHS must have the same type; result has the same type as both sides.
−	float	float	float	
−	double	double	double	
−	period	period	period	
−	time	period	time	
*	int	int	int	LHS and RHS must have the same type; result has the same type as both sides.
*	int	int	int	LHS and RHS must have the same type; result has the same type as both sides.
*	float	float	float	
*	double	double	double	
*	period	int64	period	Multiplication of the period value by the int64 value.
/	int	int	int	LHS and RHS must have the same type; result has the same type as both sides. Integer division.
/	float	float	float	
/	double	double	double	
/	period	int64	period	Integer division of period value by the int64 value.
%	int	int	int	LHS and RHS must have the same type; result has the same type as both sides.
%	period	period	period	

[1] Left-hand side data type; [2] Right-hand side data type; [3] int stands for any of the four integer data types.

Table B.1: Artithmetic operators and expressions.

`is null` and `is not null` are unary operators applicable to optional data types. Some examples are presented below:

```
1 int32? a = null |: int32;
2 int32? b = 4;
3 bool is_a_null = (a is null);
4 bool is_b_not_null = (b is not null);
```

Listing B.27: Examples of tests for nullability.

Boolean Expressions Boolean expressions in CALL are the standard *and*, *or* and *not* boolean operations. The following listing shows some examples:

```
1 bool t = true;
2 bool f = false;
3 bool also_f = t and f;
4 bool also_t = t or f;
5 bool also_also_f = not t;
```

Listing B.28: Examples of boolean operators.

B.2.4 Collection Expressions

An operator applicable to all collection types is the *count* operator. This operator obtains the number of elements in the collection The following example demonstrates the count operator:

```
1 set<int32> s = { 1, 2 };
2 int32 two = #s;
```

Listing B.29: Count operator.

Memberhip in sets and bags can be tested using the `in` and `not in` operators. The following examples demonstrate the operators:

```
1 set<int32> s = { 1, 2 };
2 int32 two = 2;
3 int32 three = 3;
4 bool is_true = (two in s);
5 bool is_also_true = (three not in s);
```

Listing B.30: Collection membership operators.

Set operators (union, intersection and minus) can be applied to sets and bags. The following examples demonstrate these operators:

```
1 set<int32> s1 = { 1, 2 };
2 set<int32> s2 = { 2, 3 };
3
4 set<int32> one_two_three = s1 union s2;
5 set<int32> two = s1 intersect s2;
6 set<int32> three = s2 \ s1;
```

Listing B.31: Set operators.

A useful operator for sets, lists and bags is the `the_one`. This operator extracts the single element in the set or bag (a runtime error will occur if more than one element are in the set or bag or if the set or bag is empty).

```
set<int32> set_with_one = { 1 };
int32 one = the_one set_with_one;
```

Listing B.32: Single element extraction.

CALL also supports comprehension operators. These operators can be used to create collections from other collections. These are applicable to sets, lists and bags and transforming a collection into a different one. The comprehension operator starts with the `collect` keyword and the rest of definition between braces: `collect{<collect contents>}`. The collection contents are split into two or three sections. The first section defines the collection source, the last the collection contents and an optional middle one, the filter.

A comprehension without a filter looks like `collect{<source> @ <contents>}` and a comprehension with a filter looks like `collect{<source> | <filter> @ <contents>}`. The source part defines the collection type and the elements to iterate over to build a new collection. The syntax for the source part is `<type> from <var> [, <var> [, ...]] in <col> [; <var> in <col> [; ...]]`. `<type>` is the type of the resulting collection, `<var>` are names of variables that will be local to the comprehension (and usable in the `<contents>` part). `<col>` are collections the values of variables are drawn from.

The filter, if it exists, is a boolean expression. The contents expression of the comprehension operator evaluates to the target collection type.

The operator works by creating a new collection and iterating over all collections assigning values to local variables. For each assignment, the filter, if defined, is evaluated. If the filter returns `true` or if there is no filter, the contents part of the comprehension is evaluated and the result of the valuation added to the collection. The following examples demonstrate the comprehension operation:

```
set<int32> set1 = { 1, 2 };
set<int32> set2 = collect{set<int32> from a in set1 @ a + 1};
// set2 == { 2, 3 }

set<int32> set3 = collect{set<int32> from a, b in set1 @ a + b};
// set3 == { 2, 3, 4 }  (1 + 1, 1 + 2, 2 + 1, 2 + 2)

bag<int32> bag1 = collect{bag<int32> from a, b in set1 @ a + b};
// set1 == { 2, 3, 3, 4 }

set<int32> set4 = collect{set<int32> from a in set1 | a % 2 == 0 @ a };
// set4 = { 2  };

list<int32> list1 = [ 1, 2, 3, 4 ];
list<int32> list2 = collect{list<int32> from a in list1 | a > 2 @ a - 2 };
// list2 == [ 2, 3 ]
```

Listing B.33: Comprehension operators.

B.2.5 Quantifier Expressions

A quantifier expression is an expression that evaluates to a boolean and corresponds semantically to one of the qualifiers $\forall$ or $\exists$. Quantifier expressions start with the `forall` or `exists` keyword and then contain a specification of bind variables as with comprehension operators, described in Section B.2.4.

```
set<int32> set1 = { 1, 2 };
set<int32> set2 = { 2, 3 };
bool is_true_1 = (forall i in set1 @ i < 3);
bool is_false_1 = (forall i in set1 @ i > 1);
bool is_true_2 = (exists i in set2 @ i < 3);
bool is_false_2 = (exists i in set2 @ i < 2);

bool is_true_3 = (forall i in set1; j in set2 @ i <= j);
bool is_false_3 = (exists i, j in set2 @ i == j * 2);
```

Listing B.34: Quantifier expressions.

B.2.6 Field Access Expressions

There are two main field access expressions. Those that access the value of a `typedef`, or the current instance in a `class`, and those that access a field in a `struct` or `class`.

The first type of field access expressions use the `this` keyword.

```
typedef nonzero restricted int32 {
    invariant zero_no_allowed { this != 0 };
}

class person {
    int32 age;

    expr bool is_toddler() {
        return this.age >= 1 and this.age <= 3;
    }
}
```

Listing B.35: Field access expression: `this` keyword.

The second type of field access expressions accesses a field in a `struct` (Section B.1.4) or `class` (Section B.1.5). This is done using the `.` (dot) operator.

```
class person {
    int32 age;

    expr bool is_older(person p) {
        return age > p.age;
    }
}
```

Listing B.36: Field access expression: dot operator.

Note that in the example above, the left-hand side of the > operator is *not* a field access expression, but a variable expression as described in Section B.2.2. The equivalent code `this.age` would have involved two field access expressions: `this` that accesses the current object and the field access expression `.age`.

B.2.7 Function and Method Invocation Expressions

A function invocation expression is the name of a function followed by a comma-separated list of arguments enclosed in parentheses. This expression invokes the function with the given name.

```
class person {
    int32 age;

    expr bool age_greatereq(int32 v) {
        return age >= v;
    }

    expr bool at_least_18() {
        return age_greatereq(18);
    }
}
```

Listing B.37: Function expression.

A method invocation expression is similar to a function invocation, but is preceeded by the dot operator.

```
class person {
    int32 age;

    expr int32 my_age() {
        return age;
    }

    expr bool older_than(person p) {
        return age > p.my_age();
    }
}
```

Listing B.38: Method expression.

B.2.8 Casting and conversion

A *cast* is a compile-time operation with run-time verification, where an expression of a given type is re-interpreted as if it had a different type. Casting does not change the involved object in any way.

When an expression of type `A` is cast into type `B`, it behaves as type `B` in all subsequent expressions. At run-time it is checked that the cast can be performed: if the actual type of the object is `B` or a sub-type of `B`.

Casting is done using the `cast` operator.

```
struct a { int32 x; }
struct b : a { int32 y; }

b v;
a w = cast<a>(v); // OK
a x = v;          // OK, due to auto-casting.
b y = x;          // Compilation error.
b z = cast<b>(x); // OK, but run-time checked.

class c {
    expr int32 foo1(a val) {
        return val.y; // compilation error
    }

    expr int32 foo2(a val) {
        return cast<b>(val).y; // OK, assuming val is of type b
    }
}
```

Listing B.39: Casting.

Casts in CALL are only used between types and sub-types. A cast to a supertype is always allowed and is added implicitly by the compiler. A cast to subtype is always explicit.

A conversion is performed when a data type needs to be coerced into a different type, but the two types are not sub-types. In that case, the value of the expression itself must change and a new value is used. Conversion happens, for example, between primitive types.

Consider the example below:

```
int32 a = 5;
int64 b = convert<int64>(a);
int64 c = a;              // Same as above, works due to auto-conversion
int64 d = cast<int64>(a); // Does not compile
```

Listing B.40: Conversion.

Because `a` is an `int32`, it cannot be cast or assigned to an `int64`. However, CALL knows how to convert an `int32` value into an `int64`.

CALL will automatically convert:

- `int8` to `int16`;
- `int16` to `int32`;
- `int32` to `int64`;
- `int64` to `float`;
- `float` to `double`;
- `ascii` to `string`;
- Any type `T` to `T?`;
- Any type `T` to `U?`, if `T` can be converted to `U` (this is a more general version of the previous rule);
- `set<T>` to `set<U>`, if `T` can be converted to `U`;
- `list<T>` to `list<U>`, if `T` can be converted to `U`;
- `bag<T>` to `bag<U>`, if `T` can be converted to `U`;

CALL will also automatically chain conversions as needed. So, for example, `set<int16?>` can be automatically converted into a `set<int32?>?`, by converting the set of `int16?` into a set of `int32?` and then applying optional conversion.

CALL allows explicit conversion to remove optionality from a type. This allows expressions such as:

```
int32? a = 5;                // Automatic conversion to optional
int32 b = convert<int32>(a); // Explicit conversion from optional
```

Listing B.41: Explicit optional removal.

B.2.9 Operator Precedence and Grouping

Operators in CALL have the following precedence:

1. `the_one`, `#`, `cast`, `convert`, function invocation, field access
2. `*`, `/`, `%`, `intersect`, `\`
3. `+`, `-`, `union`
4. `==`, `!=`, `<`, `<=`, `>`, `>=`, `in`, `not in`
5. `is null`, `is not null`
6. `and`
7. `or`
8. `forall`, `exists`

Expressions can be grouped using parentheses as shown in several examples. This construct allows overriding operator precedence.

B.2.10 Unification

Unification is the process by which the CALL compiler tries to match an expression to a specified data type, or tries to find a common data type between two expressions. Unification is triggered in two ways:

- Need to assign an expression to a data type. For example:

```
1 A foo;
2 B bar = foo; // Need to assign foo to type B
```

Listing B.42: Unification casting.

This type of unification requires checking whether there is an automatic path of casts and conversions of the expression that would result in the provided type.

- Need to find a common type between expressions. For example:

```
1 A foo;
2 B bar;
3 C c = { foo, bar };
```

Listing B.43: Unification of expressions.

Here, a unification is used to find a common type of `foo` and `bar`. This type is used for the right-hand set. If, for example, both `A` and `B` are subclasses of `S`, then the expression { `foo, bar` } would have type `set<S>`. Then, through unification casting, the expression would be converted or cast into type `C`.

B.3 Operations

B.3.1 Operable Types

An *operable type* is a data type that can define operations. In CALL there are two types of operable types: typedefs and classes. An operable type contains both state and operations, much like a

class in an object-oriented language. These were described in more detail in Section B.1.5. Operable types can contain both invariants and operation, defined in Section B.3.2 and Section B.3, respectively.

B.3.2 Invariants

An invariant is a boolean expression that must be true for a value to be a valid instance of the type. Invariants are named to allow runtime reporting of errors, but the names are otherwise not used.

Invariants are not held when the object is mutating, that is, when a method is being executed. For example, consider the following typedef:

```
typedef even_set restricted int<32> {
    invariant always_even { #this \% 2 == 0 }

    void add_numbers(int32 n0, int32 n1) {
        this = this union \{ n0 \};
        this = this union \{ n1 \};
    }
}
```

Listing B.44: Object mutation and invariants.

If an `even_set` is empty and `add_numbers` is invoked with arguments `1` and `2`, then the invariant holds before the method is invoked and after the method has been invoked. However, in the middle, after `this = this union { n0 };`, we will have `#this % 2` evaluating to `1`, breaking the invariant. However, this is OK because we're in the middle of a mutation.

A mutation starts when a method is invoked on an object and ends when the method ends. Note that nesting does not affect mutation.

B.3.3 Operations

Operations are declared as methods in an operable type. There are two types of operations: single-expression and turing-complete. A single-expression operation consists of a single expression. It cannot have loops or conditional statements and it cannot mutate the object it is being invoke on. Single-expression operations can call other operations, but only single-expression operations. Single-expression operations are marked with the `expr` keyword.

Turing-complete operations are the default. They contain none of the restrictions mentioned above for single-expression operations.

```
typedef number restricting int32 {

    // Single-expression operation
    expr number next() {
        return convert<number>(this + 1);
    }

    // Turing-complete operation
    bool round_near_10() {
        int32 remain = this % 10;
        int32 result = this - remain;

        if (remain >= 5) {
```

```
14              result = result + 10;
15          }
16
17          this = convert<number>(result);
18          return remain == 0;
19      }
20  }
```

Listing B.45: Single-expression and turing-complete operations.

Operations are built with several types of statements:

- Variable declarations. These have the form `type name = value;` and declare a variable with name `name` of type `type` and initialized to the value of expression `value`.
- Variable assignment. These have the form `name = value;` and assigns the value of expression `value` to the variable with name `name`.
- Return statement. These have the form `return value;` and end the current method returning the value defined by the expression `value`.
- If statement. These have the form `if (bexpr) { ifcode }`, where `bexpr` is a boolean expression and `ifcode` a sequence of statements. They may also have the form `if (bexpr) { ifcode } else { elsecode }` where `elsecode` is a sequence of statements.