Álvaro António Silva Ferreira Vieira dos Santos

Bachelor of Science

Distributed Live Programs as Distributed Live Data

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics

Adviser: João Ricardo Viegas da Costa Seco, Assistant Professor, NOVA University of Lisbon

Co-adviser: João Carlos Antunes Leitão, Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2019

Abstract

In a distributed system, reasoning about the impact that modifying one system component may have on other components is non-trivial — particularly, in systems where components are highly interconnected or numerous (e.g., in a distributed system following a microservices architecture). Reasoning about the effects of these modifications may involve large amounts of bookkeeping (e.g., to keep track of component interdependencies), and if the system evolves as a reasoning process is ongoing, the results of that process may end up being outdated and ultimately useless. Reconfiguring a running system may also require parts of it to be temporarily made unavailable, in order to guarantee that components do not see inconsistent intermediate states of the system.

Live programming blurs the line between developing a program and executing it, by continuously providing feedback to programmers as they develop their programs — allowing for the constant evolution of their programs without halting their execution. This dissertation explores the application of ideas underpinning live programming to the design and implementation of distributed systems. Our approach is based on a programming model that allows the construction and upgrade of distributed systems without impacting their availability, correctness, or efficiency.

Keywords: distributed systems, distributed system reconfiguration, live programming, continuous feedback, programming languages, distributed system evolution

Resumo

Num sistema distribuído, raciocinar sobre o impacto que modificar uma componente do sistema pode ter sobre outras componentes não é trivial — particularmente em sistemas cujas componentes são fortemente interconectadas ou numerosas (e.g., num sistema distribuído construído segundo uma arquitectura de microserviços). Raciocinar sobre os efeitos destas modificações pode requerer que o sistema mantenha um grande volume de dados (e.g., para ser possível conhecer as interdependências entre os componentes do sistema), e se o sistema evoluir enquanto está a ser feito um raciocínio, as conclusões alcançadas por esse raciocínio podem estar imediatamente desactualizadas, acabando por ser inúteis. Reconfigurar um sistema em execução também pode implicar que partes do sistema fiquem temporariamente indisponíveis, para que seja possível garantir que nenhuma componente observa um estado intermédio e incoerente do sistema.

A abordagem *live programming* atenua a distinção entre as fases de concepção e execução de um programa, fornecendo ao programador *feedback* continuo durante o desenvolvimento dos seus programas — possibilitando que os programas sejam modificados sem que a sua execução seja suspensa. Esta tese explora os conceitos subjacentes a *live programming* e a sua aplicação à modelação e implementação de sistemas distribuídos. A nossa abordagem tem por base um modelo de programação que permite a construção e evolução de sistemas distribuídos sem pôr em causa a sua disponibilidade, correcção, ou eficiência.

Palavras-chave: sistemas distribuídos, reconfiguração de sistemas distribuídos, live programming, feedback contínuo, linguagens de programação, evolução de sistemas distribuídos

Contents

Chapter 1

Introduction

1.1 Context

Live programming is somewhere between a philosophy on how to interact with a computer process, and a programming language/environment implementation technique. At its core, it consists of giving the programmer immediate feedback on how their edits will affect the program's behaviour, without interfering with its execution if it happens to be running as the edits are being performed [21, 26]. But such a definition is (intentionally) loose, and even things as (deceptively) simple as an Integrated Development System's (IDE) automatic syntax verifications could be argued to be a type of live programming environment: as the user writes their program, they are immediately given feedback on whether it is syntactically valid or not — which, obviously, affects the program's behaviour (an invalid program will be fully or partially rejected by the compiler/interpreter). Thus, whether we consider a programming language or environment to be live depends on the context.

As an example, consider a highly dynamic programming language[1] like Python [31], used to compose a program in a REPL[2]. Besides immediately trying to evaluate any expression the programmer inputs, a REPL gives them the ability to inspect the properties of their program's artifacts in an exploratory fashion — through constructs such as `__dict__` [30], which lists all the attributes of an object, or `exec`, `eval`, `compile`, and `str`, which blur the line between code and data by allowing the conversion between each other [29]). Contrast this with the traditional 3 step modify-compile-run development cycle

[1] In this context, "dynamic programming language" should be understood not as necessarily referring to a dynamically typed language, but to the notion of dynamic introduced in [21] - i.e., a language that promotes the use of high-level programming constructs, such as higher-order functions, and focuses on facilitating the development of programs over their runtime performance.

[2] REPL stands for Read-Eval-Print-Loop; The Python documentation refers to it as "the Python Interpreter" [28].

for a language like C, and it becomes obvious that the Python development experience is more in line with the goals of live programming than C's, even though editing Python in a REPL is not as live as other languages created specifically to perform live programming (e.g., the programmer must fully complete a function definition before being able to get any feedback about the environment over which the function closes at runtime[3] — and even then, this must be done *manually* through the insertion of `print` statements or the use of a debugger).

Thus, it could be argued that the core goals of live programming date to the creation of the first interpreters for the LISP language family, or the creation of the first IDEs for Smalltalk — meaning that there's up to over 60 years' [4] worth of ideas and terminology to unpack in order to have an accurate overview of what live programming actually is, what its impacts have been on computer science as a whole, and what work is left to be done with the term. Previous work has documented the evolution of live programming [40] and live coding[5] [44], as well as tangentially defined some of the terms that relate to live programming[6] [39]. These accounts of live programming lack in breadth — although justifiably, given the broadness of the concept —, and are either shallow and short, or more involved but extremely lengthy. Due to this, we choose to present our own account a few of the terms related to live programming (mostly those found in [39]) here. We use subscripts to denote alternative spellings present in the literature.

- **Live coding**$_{\text{Livecoding}}$ is a contemporaneous term that subclasses live programming: it refers specifically to the usage of live programming in order to perform for an audience in real-time. It seems that most of the efforts related to live coding pertain to the creation of tools for musical performances, such as Impromptu [37, 38] and Sonic Pi [36], although the term has also been used to described languages for creating graphical environments, such as Fluxus [9]. The term seems to be the most popular of the ones we present here (perhaps surpassing live programming itself), and organisations/conferences such as TOPLAP [42] and ICLC [15], respectively, exist specifically to foster the usage and development of these live coding tools.

- **Live editing** is found in the literature, albeit its usage is usually in the context of some other term (e.g., describing a feature of a live programming environment to be "live editing"), and it refers to the actual action of editing a system as it runs without halting its execution but while still causing it to generate feedback on those edits. One usage of the term appears in the paper introducing Morphic [18], describing an user interface builder that allows widgets to be edited (e.g., by moving their position, by having their colour changed) without stopping the execution of the

[3] Note that determining the environment over which an arbitrary python function closes is not trivial, since it can be changed programatically at runtime through the `exec` statement [29].

[4] As a reference point, consider that LISP's development started in the late 1950s [19].

[5] Live coding is a variant of live programming which we introduce shortly.

[6] We explicitly classify the definitions as "tangential" because they were not the main focus of the paper.

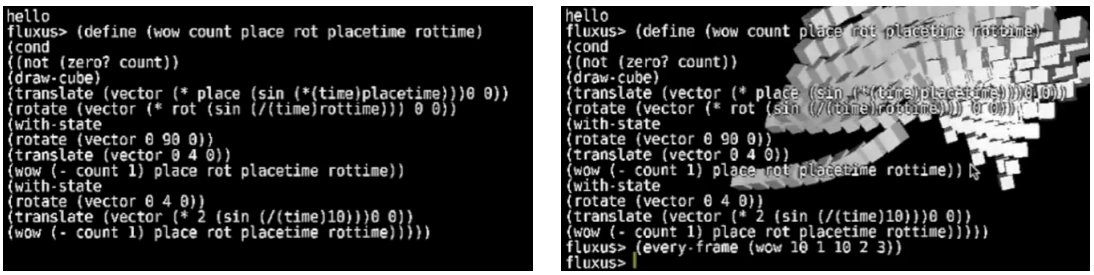

Figure 1.1: An example of live coding in Fluxus [35]: as soon as `(every-frame (wow 10 1 10 2 3))` is input, the environment starts rendering the graphics, which the user can then interact with by clicking and dragging with their mouse cursor.

system (i.e., the system will still respond to user input events, as it would if no edits were being performed). Other mentions of the term can be found in [21], [40], and [13, 14].

- **Just-in-time**$_{\text{Just in time}}$ **programming** is a term coined by Richard Potter, who describes it as "the implementing of algorithms during task-time, (the time when the user is actually trying to accomplish the task), [...] to allow users to profit from their task-time algorithmic insights by programming"[7] [27].

  Swift et al. claim that just-in-time programming is a broader definition of live programming than "live programming", "live coding", or "cyber-physical systems" [39]. We find this claim confusing: both due to the implication that just-in-time programming is broader than something it is classed as ("live programming"), and because we understand just-in-time programming as defined by Potter to be a methodology for developing algorithms regardless of the specific language or environment being used[8], but understand live programming as being an inextricable characteristic of the tools being used to write a program and ultimately amenable to any software development philosophy — even if the immediate feedback afforded by live programming may be particularly conducive to development philosophies like just-in-time programming.

  The term appears sparsely throughout the literature, and to the credit of Swift et al., its meaning seems to have partially converged towards being a synonym of live programming over the years [5, 16, 23, 32], even if the older meaning is still used in some recent works [38].

- **Cyber-physical**$_{\text{Cyberphysical}}$ **programming** is among the rarest terms in this list[9]. Of

[7] To preserve the readability of the modified quote, we have signalled our edits by underlining them, instead of placing them in square brackets as usual.

[8] Which is partially supported by Potter's own words: "Any programming system could conceivably be used for just-in-time programming, including C, PASCAL, keyboard macros, scripting languages, or PBD" [27].

[9] In total, we have found only 4 usages of the term across the ACM Digital Library, the IEEE Xplore Digital Library, and Google Scholar: [1, 3, 38, 39] (excluding those that only use the term to cite one of the articles whose title contains it).

> the usages we found, Atkins seems to use the term as synonymous with the act of programming a cyber-physical system [1] — a system that integrates virtual computational processes with physical processes [8] —, without any indication of associating the term with live programming, while Campusano et al. [3] simple paraphrase Swift et al. [39], who in turn summarises the ideas discussed in the work of Sorensen and Gardner [38] (which coined the term). The concept, as defined originally, means to program a cyber-physical system in real time (i.e. use live programming to control a cyber-physical system)[10].

In short: despite the plethora of terms related to live programming and the age of various concepts that could be classed under this umbrella of terms, it seems to have remained a niche topic until recently, with much of the related work falling under the scope of artistic performances, via organisations such as TOPLAP or the ICLC, or under the scope of pedagogical efforts, via the creation of visual and textual introductory programming languages such as Scratch and Alvis LIVE!, respectively. Although laudable in their own right, we find the research community in general not having taken a more explicit[11] interest in this area to be a missed opportunity for innovation. However, there does seem to be a growing interest in using live programming for more "serious" efforts over recent years: in 2016, 2017, and 2018, there has been a workshop focused solely on Live Programming co-hosted with ECOOP and SPLASH (in 2016 and 2017/2018, respectively) [17], and a paper on a live programming environment was even presented at POPL '19 [26].

1.2 Problem Statement

As described in §1.1, live programming has enjoyed limited application outside of niche areas such as live coding. Thus, unsurprisingly, of what little research has been done on how live programming and distributed systems may be integrated together, most of it has been in the context of live coding — specifically in the context of allowing for networked musical live coding through programming languages/environments such as Sonic Pi [36], extramuros [24], and Impromptu [37]. While these works make use of distributed systems in a live programming context, they do so without any care for the distributed system itself as an object of study — i.e., they care only about providing a platform for enabling multiple people to live code together, which happens to be implemented as a distributed system.

We believe that another view is possible: we consider that a distributed system is in itself a live software artifact, whose state evolves as new events cause processes to message

[10] It should be noted that [3, 39] characterise it as an extension of live *coding*, although the original coinage of the term does not have the connotation of usage for performances that live coding carries.

[11] We quantify it as explicit because live programming does end up showing itself in all kinds of topics, even if only at a primitive level — e.g., in topics regarding reflection, interpreters

others in the network[12], and whose lack of availability is often undesirable or outright unacceptable. We posit that reasoning about these system's properties (e.g., replication and fault tolerance, availability, consistency, and other such typically relevant properties of distributed systems) may be interesting to do through the ideas of live programming — ideas which stress the importance of being able to edit programs as they run, with immediate feedback on how these edits are affecting these programs, done so without requiring that they be halted.

Thus, we enunciate the question this dissertation wishes to explore and, ultimately, solve:

Motivating Problem

> ***C**an the design, deployment, update/evolution and maintenance of distributed systems be modelled using a programming language theory approach based on the fundamental ideas and goals of live programming? If possible to model a distributed system like this, is it useful to do so (i.e., does the implementation of that model provide any practical, useful, efficient tools to deal with distributed systems)?*

Specifically, we are interested in dealing with questions such as "Can this system allow for one of its components to change while guaranteeing that all messages being exchanged in the system will still behave appropriately?" and "What can we conclude about the impacts that changing a local component will have on the system as a whole?", which map directly to "traditional" live programming questions such as "Can we allow this object's interface to change without impacting running code that depends on it?" and "How will changing a line of code impact the results of the program's execution?".

However, it must be noted that the problems of designing, deploying, updating and maintaining distributed systems are not novel — after all, the distributed systems community has existed for decades at this point, and these issues obviously arise in its work. What we claim is not that no one has ever looked at these problems, but rather, that few attempts have been made at attempting to solve these problems specifically through the lens of live programming. In §2 we look at previous work which has tried to do so, even if not enunciating it explicitly as "using live programming to work on distributed systems". Examples of such work are those of Rosa et al. [33], Rodrigues et al. [22], and Frömmgen et al. [10], which automatically adapt a system's structure or algorithms at runtime based on its current environment (e.g., based on network performance or cpu utilisation), or that of Seco et al. [34], which consists of a programming model which allows the specification of microservice's interfaces and their evolution in a safe way (i.e., one that allows services to change their definitions without breaking all of the services that depend on it or requiring them all to be upgraded manually).

[12] Which may trigger new messages, who themselves trigger new events, which in turn trigger more messages, which [...].

We propose that to address the problem of using live programming notions in the context of distributed systems, we must first introduce 3 definitions that describe different ways of integrating the two notions:

LLP: Definition

Local Live Programming *corresponds to the usual definition of live programming found in the literature, which was given at the start of this chapter (1.1). In the context of a distributed system, having local live programming means that (at least some of the) processes in the system allow for live programming to take place relative to their own local data, without any regard for the existence of other processes in the system.*

ILP: Definition

Interprocess Live Programming *corresponds to live programming in a context where multiple processes must communicate with each other, via some mechanism with appreciable delays in communication (such as a networked set of processes). This corresponds to the notions of distributed/networked live coding previously presented.*

DLP: Definition

Distributed Live Programming *corresponds to using live programming to build and reason about a set of networked processes, by monitoring their evolution to ensure it can only occur safely (i.e. in a way that does not disrupt the system's availability or correctness). To the best of our knowledge, this is a novel notion.*

While both ILP and DLP encode some notion of the existence of a distributed system, the key difference between them is that ILP merely takes the distributed system as a fact of the world, and strives to hide its existence and impact on the program (i.e., ideally, ILP would have a set of running processes be indistinguishable from a single process performing all their functions locally), while DLP reifies the attributes of the distributed system in the programming model, allowing the model's implementation (i.e., the programming language/environment) to expose the properties of the distributed system to the running program itself.

Our approach, described in §3, can then be summarised as using DLP to attempt to solve the Motivating Problem, through the creation of a programming model to capture the properties of distributed systems, and its implementation.

1.3 Document Structure

The remainder of this document is organised as follows:

- **Chapter 2:** Here, we discuss work related to live programming and its sister subjects (as discussed in §1.1). We divide the discussion according to the areas of research the work primarily falls into.
- **Chapter 3:** Here, we discuss how we plan to attack the problem of integrating the ideas of live programming into distributed systems, and discuss what tangible results we expect to have as a result of this dissertation.
- **Chapter 4:** Here, we present the work plan we intend to follow during this research, including at which point we plan to have some of the deliverables spoken of in §3.2.

Chapter 2

Related Work

The problem we aim to solve is not new: designing, deploying, updating, and maintaining distributed systems intersects the interests of several research communities, such as those of software engineering, distributed systems, and programming languages. Each of these communities has partially contributed in their own way to solving this problem, through the creation of tools such as software design methodologies like service-oriented architectures and event-driven architectures, implementations of systems such as Apache Kafka and Apache Zookeeper, and formal tools like process calculi.

Despite our problem not being new, we believe that our approach is unique in its treatment of distributed systems as live software artifacts, whose characteristics are treated as first-class citizens of their programming language's semantics — that is, the language itself must constrain what programs may be written locally so that their global effects (i.e., effects on the other nodes) do not cause the system to misbehave, in the same way that a statically typed language will not allow some programs to be run in order to prevent runtime errors.

Due to the uniqueness of our approach of trying to use live programming as a tool for reaosning about distributed systems, and the broadness of our problem, it is hard to accurately gauge what previous work relates to our dissertation. Below, we present only the most directly relevant prior works that we have found, including those that, despite not trying to solve the same problem, may provide useful building blocks to do so.

2.1 Programming Language Based Work

2.1.1 Contracts

Contracts are a "description of the external, observable behavior of a service" [4] — or, in other words, the interface a component of a system exposes to the other components.

As described in [4], contracts can be seen as the composition of write actions and read actions through 3 binary operators: prefixing, meaning the execution of an action before another; external choice, meaning that the service's client may choose which action to take next; and internal choice, meaning that the service itself will decide one of options next. Naturally, the semantics of write and read actions depends on the specific details of the underlying system. The authors of [4] also note that contracts as thus described are simply a restricted form of behaviour types (which themselves relate to broader concepts of process calculi).

This dissertation builds directly on previous work [34] which describes a programming language designed to allow for the safe evolution of components in a microservices architecture. In this work, "evolution" corresponds to performing changes to the contracts exposed by the components, and "safe" corresponds to the notion of only allowing these changes to take place (i.e., be deployed/integrated into the running system) if they extend the previous contract (e.g., by adding new definitions to a component), or reduce it only if no other component depends on the definitions being removed/altered in a breaking way (e.g., the removal of fields from a record type's definition). The changes, if allowed, require no further programmer input; Code adapters are generated at the components that consume other components' definitions in order to convert between the old and new definitions.

Although not directly mentioning contracts, various notions of compatibility between services are explored in [2], which may be interesting to explore in conjunction with the other works mentioned in this subsection.

While serving as the basis for this dissertation's development and encoding some notion of evolution or replacement of existing services, some these previous works are limited either by their inability to deal with the "real world": they do not account for practical implementation considerations such as component faults or their replications, or for the practical impacts that maintaining the information needed to know which services depend on each other may have on the latency in handling a request.

2.1.2 Live Programming

As described in section 1.1, a live programming language/environment gives the programmer immediate feedback on the state of the program and how their edits will affect it. Previous work in this area includes the creation of various domain-specific programming languages and tools, of which we have as examples: introducing programming concepts to children [13]; reducing the difficulty of composing components in systems following an MVC architecture [20, 21]; providing an environment for the creation of music through programming primitives [36]; the evolution of data-centric programs with immediate feedback on how the updated programs interact with the data [7]; and a programming language editor which both prevents users from inputting syntactically incorrect expressions and gives them feedback on how to complete their programs if expressions are not

deemed to be complete [25, 26].

The generality of the ideas underlying live programming makes it hard to understand which of the prior literature is actually relevant to us, as most of the work in the area has a similar-yet-different view of what is live programming. Of the examples mentioned above, we find Sonic Pi and the language of [7] to be most relevant. In Sonic Pi, the goal is not directly related to the construction of distributed systems, but rather to the construction of live musical programs that communicate via the network. What we find relevant about Sonic Pi is how it manages to support an extremely latency-sensitive experience (music) over the network in a live manner, and we are interested in whether this uses any novel techniques that can be adapted to support our goals. In the work of [7], the typechecker validating a program implies that the program is safe and responsive, which is the kind of guarantee that we want to have in our own programming language.

The lack of a single set of tangible goals shared by all works classed as "live programming" makes it hard to speak about the term's shortcomings in the general sense. Nevertheless, we find that the literature is pervasively populated with systems that do not at all care for performance, even beyond what is acceptable for a prototype intending to show the viability of using live programming in realistic contexts — for example, in [21], the authors report the creation a small Pac-Man-like game in their language. In this simple example, creating each ghost takes 0.5 seconds, and in some cases, the user will experience several seconds of latency before getting feedback on their input — effectively making the system temporarily non-live. The authors of [21] also raise another question which applies to live programming as a whole: how can a live programming system know what feedback is useful to the programmer and what feedback just "gets in the way" (e.g., predictive text completion forcing the programmer to either select one of the predicted words or to take extra steps to stop the text completion mode)?

2.1.3 Incremental Computation

Incremental computations are computations that are performed repeatedly on repeated instances of inputs that have little difference between them [43]. Incremental computations show up naturally in programs that respond to iteratively built user input: IDEs and spreadsheet manipulation programs are prime examples of this, as some of their functionalities — such as code completion suggestions or mathematical formula computation — run on input that is usually built by taking the previous version of the input and slightly altering it. The main advantage of incremental computations is that by exploiting the minute differences between consequent inputs to preserve computational effort whose output does not change the different input versions, the cost of producing an output for a new input is less than re-executing the entire program with the new input.

In particular, we are interested in the INC language [43], as it not only allows for incremental computations, but also has a very interesting property: INC programs are implemented as a network of processes sending messages to each other, with the processes

corresponding to functions (as written by the programmer) and messages corresponding to inputs and outputs. The reason for our interest in this property is how it suggests a connection to distributed systems, since they are, by definition, networks of processes sending messages to each other. However, INC is limited, in that it lacks recursion and general loop constructs.

Incremental computations are also interesting in how they naturally complement live programming, as both subjects work with input that is likely similar to whatever input preceded it, and in how they address one of the concerns with most of the other programming language theory works herein listed: the lack of consideration towards efficiency that most formalisations suffer from.

2.1.4 Dynamic Software Updates

As defined in [12], a system that allows for dynamic software updating (DSU) is one that allows for the arbitrary modification of a running program[1]. Although superficially similar, DSU and live programming are not the same. In fact, DSU and live programming can be seen as complementary: live programming gives the programmer immediate feedback about their modifications without interfering with the program's execution, and DSU can be used as the mechanism which allows for those modifications to be applied to the program without disrupting its execution.

The notion of updating a system without disrupting its execution is well-established, with multiple prior approaches on how to do so — as described in Hicks' thesis [12]'s. These approaches include: maintaining replicas of the hardware which runs the program, deploying the updated program on a replica, modifying the replica's state as needed based on the primary version's state, and finally setting the replica as the new primary; identifying what state a process uses and converting it into the state the updated process requires; and dynamically linking and unlinking code at runtime to manage what symbols the program may access. These approaches are all described in Hicks' thesis.

In [41], the authors note that using DSUs in the context of developing a program in a live programming environment requires different guarantees than using DSUs in the context of evolving and maintaining a long running deployed system, and try to reconcile both of these usages. This ties directly to the problem this dissertation aims to solve, as distributed systems, once deployed, are typically meant to be long running.

[1] Of course that allowing for truly *arbitrary* modification is not feasible in a system that wishes to provide any sort of safety guarantees, as guaranteeing safety necessarily implies that some updates will be disallowed in order to stop the system from misbehaving.

2.2 Distributed Systems Work

2.2.1 Runtime Support Systems

A distributed system is, by its very nature, embedded in a dynamic environment: nodes and communication links may become delayed for arbitrary amounts of time, or even fail. To deal with this, the distributed systems community has developed systems that are capable of automatically responding to the conditions of their environment, such as the CPU usage of their nodes or the median time the system is taking to process incoming requests. This ties to our notion that a distributed system is a live artifact whose properties should be known to the processes which compose the system, allowing the system as a whole to be reasoned about as a first-class entity (i.e., in the same manner that the programmer may create a set of integers and reason about its typical properties — size, membership, etc), and not simply being an opaque concept on which computation happens. As examples of work in this area, we discuss the works in references [10, 22, 33].

2.2.1.1 Self-Management of Adaptable Component-Based Applications

In [33], the authors develop a system based on components and performance indicators. In their system, the programmer is responsible for creating components, each of which may have multiple implementations — which the authors call adaptations —, and goals. Goals encode different states that the system should converge to, and adaptations encode the behaviour that the system will display given when in a certain state. Adaptations must describe what actions they perform, what are their requirements (what pre-conditions the adaptation requires the system be in), what are their impacts (how they affect the system's performance indicators), and how long the adaptation's actions take to stabilise their impact on the system. Goals are either exact (e.g., a performance indicator must be above a threshold) or simply desirable to optimise (e.g., a performance indicator should be maximised). Their system converts the adaptations and goals into a set of rules (ensuring that rules do not try to perform incompatible actions, like simultaneously selecting adaptations that require opposing conditions), which are then evaluated at runtime to decide which adaptations to activate/deactivate, which allows the system to adapt to the status of the performance indicators.

The obvious drawbacks of this approach are related to how much of this must be done manually: the programmer must explicitly program all the alternative adaptations, design the system in such a way that it functions for every combination of adaptations that may actually be executed together, and statically estimate the impact that each adaptation will have on the performance indicators[2].

[2]However, the authors did state that there was ongoing research on how to improve the performance indicators at runtime through techniques such as reinforcement learning.

2.2.1.2 Switching Protocols at Runtime

Both of the works we discuss in this section tackle a specific instance of a more general problem. The general problem is: given a set of algorithms, all of which solve the same problem but whose efficiency is tailored for differing scenarios (e.g., a set of algorithms optimised for different ratios of reads and writes), how can a running system choose the best one for the conditions it is currently executing in?

In [10] and [22], the authors tackle an instance where the algorithms solve[3] the (Asynchronous) Total Order Broadcast[4], with one being optimised for scenarios where minimising the delay of reads is paramount, and the other being optimised for the minimisation of write delays. The formal definition of (Asynchronous) Total Order Broadcast are not directly relevant to us — it suffices to say that the problem is equivalent to getting every process in a distributed system to agree on which messages to deliver next in a stream of messages.

The work presented in [22] performs the algorithm swap by having the system decide at some point (based on the current conditions the system is operating in — similarly to the work presented in §2.2.1.1) that the current algorithm is no longer optimal and must be replaced by another one. At this point, both the old and new algorithms are used in parallel. A process keeps delivering the old algorithm messages as usual, while withholding the new algorithm's messages that it receives. Once a process receives a message from every other process agreeing that the algorithms may be commuted, it does so, by delivering the buffered messages from the new algorithm (taking care to discard those that were already received through the old algorithm), and ignoring the old algorithm from that point onwards. The work presented in [10] and detailed in [11] follows the same general principles as the one of [22].

These works are interesting for the possible research direction that they present, and the general problem that they implicitly introduce. However, these works are not aligned with our goals: they are tied to a specific problem and specific algorithms, all of which are fully known and implemented *a priori*, but our goals are more focused on offering tools to work in an environment where algorithms are constantly evolving through programmers' actions.

2.3 Miscellaneous

2.3.1 Database Schema Evolution

Many of the Database Management Systems (DBMSs) of today are specialised types of distributed systems where the ability to reason about data consistency is paramount. By

[3] As should be obvious for any reader with a background in distributed systems, the algorithms do not solve the actual Asynchronous Total Order Broadcast problem, but a relaxed version of it — the full problem is known to be unsolvable.

[4] Also known as (Asynchronous) Atomic Broadcast.

itself, this would already make DBMSs relevant to the work we'll be conducting, as there is a large body of work in the area put into exploring the guarantees afforded by different consistency models and the trade-offs required to achieve those guarantees — questions which our problem forces us to tackle, as we must manage our distributed systems in order to ensure their availability without sacrificing their correctness.

However, DBMSs are of interest to us beyond these fundamental issues. Researchers in the area of traditional relational information systems have faced problems similar to ours: specifically, how to reason about the evolution of a database's schema, with concerns not only for the schema's correctness, but also for its impacts on the system's overall performance. An example of this line of research is the PRISM Workbench [6]. This system leverages prior work on the how to invert or compose multiple schema transformations to provide a language that expresses schema changes, and functionality to evaluate the correctness and performance gains/losses of the changed schemas. This has a direct mapping to our model, where schemas correspond to component interfaces and transformations correspond to the evolution of those interfaces.

The PRISM Workbench language can be seen as a SQL-like language extended with the notion of Schema Modification Operators (SMOs). These operators encode the notions of joining, renaming, breaking up, copying, or deleting tables, and of adding, removing, or renaming columns. The SMOs are assembled into programs by the user, and then automatically used by the system to check various properties (e.g., whether the changes to the tables will cause any information to be lost, whether they introduce redundancies in the schema) and provide helpful functionality (e.g., the ability to test queries against both the old and new versions of the schema for performance comparison, the generation of an automatic rollback script to undo the deployment of the changes).

Chapter 3

Methos and Approach

3.1 Methodology

We tackle the problem of reasoning about distributed systems using the ideas of live programming with the creation of a novel programming model. Our model will be based on a set of design principles and on a set of properties — the design principles correspond to the characteristics of the environment we are trying to reason about (e.g., components may have their communications delayed due to the network's misbehaviour; components may depend on each other) and properties correspond to what we know about our system given its characteristics (e.g., a well-typed system does not break contracts by allowing a component to disrespect another component's exposed interface; a component that is allowed to evolve in a certain manner will not break the functionality of other components).

The model will, essentially, serve as a tool to reason about the interaction of components and the evolution of those component's interfaces. To design our model, we draw on the work of Seco et al. [34], and extend it with the notions of deploying and maintaining services in a replicated manner, while guaranteeing that these notions are as decentralised as possible — i.e., a "central" authority, if at all needed (to perform bookkeeping of the system's components), ought to work with the least possible amount of meta-information about the distributed system's components' interfaces. This is desirable because a centralised controller is likely to be a bottleneck, preventing the system from scaling in any useful manner.

Our model will be instantiated as a core calculus/programming language. This core calculus will define the static and dynamic semantics of our language, and will be used to prove standard properties of programming languages (term progress and term type preservation). At a high level, the static semantics of the core calculus correspond to the

types of expressions as checked by the typechecker, and the dynamic semantics correspond to the values of expressions as they are computed.

This programming language will be used to implement a proof-of-concept prototype that allows the design, deployment, and evolution of a distributed system. If possible, we will design our language to be able to interface with off-the-shelf mainstream distributed system technologies, in order to both ease our burden of implementation (through re-use of existing technologies) and to make the prototype testable with "real world" tools. The evolution of the system will be performed at the component level — that is, by replacing components for others with different behaviours.[1]

We also intend to explore the viability of imbuing our prototype with first-class notions of the properties of distributed systems (e.g., availability, type of data consistency guarantees), so that the systems implemented in the prototype language may reason about their own properties directly, and not just at a meta-level — that is, the programming language would reify the concept of "availability", giving the programmer the ability to define (at the programming language level) a subset of mission-critical components as being highly available in exchange for not guaranteeing that all of the components converge to the same state in a reasonable amount of time.

As a concrete example of what we mean in the previous paragraph, imagine a popular web service with a set of replicated "Display login HTML + CSS + Javascript" components. Such a component would, presumably, be stateless (handing off the login data to some sort of database component), and unlikely to require evolution of its logic after being created and tested in the running system. In such a scenario where their evolution is likely to be at the appearance level only (i.e., CSS and non-Javascript-breaking changes to the HTML), the programmer may find it worthwhile to mark this set of components as requiring high availability (to guarantee that no user is barred from logging into the system) in exchange for lowering its consistency guarantees upon evolution (i.e., by stating that components may evolve in an eventually consistent manner — which provides no hard boundaries on the time at which (if any) the components will all have the same appearance).

Even if viable, it is not obvious at this time whether these first-class notions would be helpful aids to the programmer. As such, even if found to be viable, we may choose to not implement these notions into our prototype.

Finally, depending on the needs we identify throughout the design and implementation of our programming model, we may explore the partial implementation of tools to aid the usage of our language, such as a structure editor similar to that of [25, 26], debugging/reflection tools for the language, or monitorization tools to inspect the status of the various components of the system as it runs. The reason we specify "the *partial* implementation" is that even just a full prototype of only one these tools would, in itself,

[1] Note that this notion is strong enough to permit the evolution of the system through changes "below" the component level — that is, to reason about a change of the code in component C_1, simply map the original version of C_1 to D_1 and the new version of C_1 to E_1, and then reason about the replacement of D_1 with E_2 (conceptually, a different component).

require an amount of time which we do not have in the context of this dissertation.

3.2 Deliverables

At the end of this dissertation, we will deliver a programming model and its formalisation, along with a proof-of-concept implementation of it. If time permits, and if we find the need to support our programming model with tools such as those specified in 3.1, we may also deliver (partial) implementations of these tools.

Additionally, we plan to produce 2 publications regarding our research, and we plan to submit our work in bridging the gap between live programming and distributed systems to the LIVE Programming Workshop 2019 edition — whose submission deadlines are usually in mid August [17] —, as this series of workshops is directly relevant to our research topic, and to the INForum symposium — whose submission deadline is the 7th of June.

Chapter 4

Work Plan

In order to achieve the goals stated in §3, we plan to develop our programming model in two separate phases — one to develop a basic model, based on the related work discussed in §2.1.1 and §2.1.2, and one to enhance that basic model with notions such of fault tolerance, replication, and other similar "real world" concerns —, as listed below:

» `23/Feb – 29/Mar [5 weeks]` Formalising the programming model based on the work presented in the work of Seco et al. [34].

» `30/Mar – 10/May [6 weeks]` Implementing the programming model.

  ▷ Exploring existing implementations of systems relevant to our goal, such as those described in §2 (e.g., implementations of systems supporting DSUs[1], such as the one described by Tesone et al. [41], implementations of systems supporting incremental computations, such as Yellin and Strom's INC language [43]).

  ▷ Using the knowledge acquired during the exploration of other systems to iteratively refine our formalisation and implement it.

» `11/May – 31/May [3 weeks]` Evaluating the technical and formal results achieved by our implementation and formalisation, respectively.

  ▷ Submitting a paper to the 2019 edition of the INForum symposium[2] describing our programming model and its implementation.

» `1/Jun – 21/Jun [3 weeks]` Describing the results of our formalisation, implementation, and evaluation.

[1] See §2.1.4.

[2] The paper submission deadline is the week after this one, on the 7th of June.

» `22/Jun – 19/Jul [4 weeks]` Iterating on our previous formalisation to generalise it, in order to tackle issues related to the replication of services (e.g., minimising the amount of information that must be known by a central coordinating authority to coordinate the various components, and controlling the dissemination of replica updates — to guarantee that no service is instantiated by 2 incompatible replicas).

» `20/Jul – 16/Aug [4 weeks]` Iterating on our previous implementation to account for our improved formalisation.

  ▷ Submitting a paper to the 2019 edition of the LIVE Programming Workshop[3] describing our efforts in bridging the gap between distributed systems and live programming.

» `17/Aug – 30/Aug [2 weeks]` Evaluating the new implementation and formalisation to compare them with the previous ones, in order to ascertain whether the benefits of generalising them are offset by an increase in the complexity of our model.

» `31/Aug – 13/Sep [2 weeks]` Describing the results of our new formalisation, implementation, and evaluation.

  ▷ Producing a paper reporting our advances relative to the previous one and submitting it to a relevant conference.

» `14/Sep – 23/Sep [1.5 weeks]` Concluding this dissertation by reviewing and presenting it.

[3] The submission deadline is not yet known, so we place this goal here based on the submission deadline of previous iterations [17].

Bibliography

[1] E. M. Atkins. "Cyber-physical aerospace: Challenges and future directions in transportation and exploration systems." In: *Proceedings of the 2006 National Science Foundation Workshop On Cyber-Physical Systems*. 2006.

[2] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. "When Are Two Web Services Compatible?" In: *Proceedings of the 5th International Conference on Technologies for E-Services*. TES'04. Toronto, Canada: Springer-Verlag, 2005, pp. 15–28. ISBN: 3-540-25049-2, 978-3-540-25049-4. DOI: 10.1007/978-3-540-31811-8_2. URL: http://dx.doi.org/10.1007/978-3-540-31811-8_2.

[3] M. Campusano and J. Fabry. "Live Robot Programming: The language, its implementation, and robot API independence." In: *Science of Computer Programming* 133 (2017), pp. 1 –19. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2016.06.002. URL: http://www.sciencedirect.com/science/article/pii/S0167642316300697.

[4] G. Castagna, N. Gesbert, and L. Padovani. "A Theory of Contracts for Web Services." In: *ACM Trans. Program. Lang. Syst.* 31.5 (July 2009), 19:1–19:61. ISSN: 0164-0925. DOI: 10.1145/1538917.1538920. URL: http://doi.acm.org/10.1145/1538917.1538920.

[5] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. "Live Coding in Laptop Performance." In: *Organised Sound* 8.3 (Dec. 2003). *NB: Organised Sound is a journal publication*, pp. 321–330. ISSN: 1355-7718. DOI: 10.1017/S135577180300030X. URL: http://dx.doi.org/10.1017/S135577180300030X.

[6] C. A. Curino, H. J. Moon, and C. Zaniolo. "Graceful Database Schema Evolution: The PRISM Workbench." In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 761–772. ISSN: 2150-8097. DOI: 10.14778/1453856.1453939. URL: http://dx.doi.org/10.14778/1453856.1453939.

[7] *Type Safe Evolution of Live Systems*. Pittsburgh, 2015.

[8] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. English. Second Edition. Cambridge, MA, USA: MIT Press, 2017. Chap. Preface, p. xi. ISBN: 978-0-262-53381-2. URL: https://ptolemy.berkeley.edu/books/leeseshia/releases/LeeSeshia_DigitalV2_2.pdf (visited on 02/17/2019).

[9] *fluxus*. Accessed 2019-02-22. URL: http://www.pawfal.org/fluxus/.

[10] A. Frömmgen, S. Haas, M. Pfannemüller, and B. Koldehofe. "Switching ZooKeeper's Consensus Protocol at Runtime." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)* (2017), pp. 81–82.

[11] S. Haas. "Runtime Adaptation for ZooKeeper." MSc Thesis. Technische Universität Darmstadt, 2016. URL: https://www.kom.tu-darmstadt.de/en/kom-multimedia-communications-lab/people/staff/alexander-froemmgen/zookeeper,.

[12] M. Hicks. "Dynamic Software Updating." Winner of the 2002 ACM SIGPLAN Doctoral Dissertation award. Doctoral dissertation. Department of Computer and Information Science, University of Pennsylvania, Aug. 2001.

[13] C. D. Hundhausen and J. L. Brown. "What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners." In: *Journal of Visual Languages & Computing* 18.1 (Feb. 2007), pp. 22–47. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2006.03.002. URL: http://dx.doi.org/10.1016/j.jvlc.2006.03.002.

[14] C. D. Hundhausen and J. L. Brown. "What You See Is What You Code: A Radically Dynamic Algorithm Visualization Development Model for Novice Learners." In: *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. VLHCC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 163–170. ISBN: 0-7695-2443-5. DOI: 10.1109/VLHCC.2005.72. URL: http://dx.doi.org/10.1109/VLHCC.2005.72.

[15] *ICLC 2019*. Accessed 2019-02-22. URL: http://iclc.livecodenetwork.org/2019/ingles.html.

[16] C. J. Light J. Drager K. *Effects of AAC systems with "just in time" programming for children with complex communication needs.* Atlanta, Georgia, USA: American Speech-Language-Hearing Association, Nov. 2012. URL: http://aac.psu.edu/?p=1281.

[17] *LIVE conference series - 2018*. Accessed 2019-02-22. URL: https://2018.splashcon.org/series/live.

[18] J. H. Maloney and R. B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment." In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 21–28. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215636. URL: http://doi.acm.org/10.1145/215585.215636.

[19] J. McCarthy. "History of Programming Languages." In: ed. by R. L. Wexelblat. New York, NY, USA: ACM, 1981. Chap. History of LISP, pp. 173–185. ISBN: 0-12-745040-8. DOI: 10.1145/800025.1198360. URL: http://doi.acm.org/10.1145/800025.1198360.

[20] S. Mcdirmid. "Component Programming with Object-oriented Signals." AAI3188517. Doctoral dissertation. Salt Lake City, UT, USA, 2005. ISBN: 0-542-31414-2.

[21] S. McDirmid. "Living It Up with a Live Programming Language." In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 623–638. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297073. URL: http://doi.acm.org/10.1145/1297027.1297073.

[22] J. Mocito and L. Rodrigues. "Run-Time Switching Between Total Order Algorithms." In: *Euro-Par 2006 Parallel Processing*. Ed. by W. E. Nagel, W. V. Walter, and W. Lehner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 582–591. ISBN: 978-3-540-37784-9.

[23] Neil C Smith. *PraxisLIVE*. Accessed 2019-02-14. URL: https://www.praxislive.org/.

[24] D. Ogborn, E. Tsabary, I. Jarvis, A. Cárdenas, and A. McLean. "Extramuros: Making Music in a Browser-Based, Language-Neutral Collaborative Live Coding Environment." In: *Proceedings of the First International Conference on Live Coding* (Leeds, UK). Available online at https://iclc.livecodenetwork.org/2015/html/100.html (accessed on 2019-02-22). Leeds, UK: ICSRiM, University of Leeds, July 2015, pp. 163–169. DOI: 10.5281/zenodo.19349. URL: https://doi.org/10.5281/zenodo.19349.

[25] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. "Hazelnut: A Bidirectionally Typed Structure Editor Calculus." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 86–99. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009900. URL: http://doi.acm.org/10.1145/3009837.3009900.

[26] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer. "Live Functional Programming with Typed Holes." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 14:1–14:32. ISSN: 2475-1421. DOI: 10.1145/3290327. URL: http://doi.acm.org/10.1145/3290327.

[27] R. Potter. "Just-in-Time Programming." In: *Watch What I Do: Programming by Demonstration*. Ed. by A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky. Accessed 2019-02-14. Cambridge, MA, USA: MIT Press, 1993. Chap. 27. ISBN: 0-262-03213-9. URL: http://acypher.com/wwid/Chapters/27JITP.html.

[28] Python Software Foundation. *2. Using the Python Interpreter — Python 3.7.2 documentation*. Accessed 2019-02-01. URL: https://docs.python.org/3/tutorial/interpreter.html.

[29] Python Software Foundation. *Built-in Functions — Python 3.7.2 documentation*. Accessed 2019-02-01. URL: https://docs.python.org/3/library/functions.html.

[30] Python Software Foundation. *Built-in Types — Python 3.7.2 documentation*. Accessed 2019-02-01. URL: https://docs.python.org/3/library/stdtypes.html.

[31] Python Software Foundation. *Welcome to Python.org*. Accessed 2019-02-01. URL: https://www.python.org/.

[32] W. R. Rohrhuber J. de Campo A. "Algorithms today: notes on language design for just in time programming." In: *Proceedings of the International Computer Music Conference*. Barcelona, Spain, 2005. URL: https://quod.lib.umich.edu/i/icmc/bbp2372.2005.135/--algorithms-today-notes-on-language-design-for-just-in-time?view=image.

[33] L. Rosa, L. Rodrigues, A. Lopes, M. Hiltunen, and R. Schlichting. "Self-Management of Adaptable Component-Based Applications." In: *IEEE Transactions on Software Engineering* 39.3 (Mar. 2013), pp. 403–421. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.29.

[34] J. C. Seco, P. Ferreira, H. Lourenço, C. Ferreira, and L. Ferrão. "Robust Contract Evolution in a TypeSafe MicroServices Architecture." English. *Unpublished*. 2019. URL: https://bit.ly/microservices_paper_techreport.

[35] Sem Shimla. *Live Coding in Fluxus*. Accessed 2019-02-22. May 2014. URL: https://www.youtube.com/watch?v=v7E6-n3M_bc.

[36] *Sonic Pi - The Live Coding Music Synth for Everyone*. Accessed 2019-02-22. URL: https://sonic-pi.net/.

[37] A. Sorensen. "Impromptu: An interactive programming environment for composition and performance." In: *Proceedings of the Australasian Computer Music Conference* (Jan. 2005).

[38] A. Sorensen and H. Gardner. "Programming with Time: Cyber-physical Programming with Impromptu." In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 822–834. ISSN: 0362-1340. DOI: 10.1145/1932682.1869526. URL: http://doi.acm.org/10.1145/1932682.1869526.

[39] B. Swift, A. Sorensen, H. Gardner, and J. Hosking. "Visual Code Annotations for Cyberphysical Programming." In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, 2013, pp. 27–30. ISBN: 978-1-4673-6265-8. URL: http://dl.acm.org/citation.cfm?id=2662726.2662734.

[40] S. L. Tanimoto. "A Perspective on the Evolution of Live Programming." In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, 2013, pp. 31–34. ISBN: 978-1-4673-6265-8. URL: http://dl.acm.org/citation.cfm?id=2662726.2662735.

[41] P. Tesone, G. Polito, N. Bouraqadi, S. Ducasse, and L. Fabresse. "Dynamic Software Update from Development to Production." In: *The Journal of Object Technology* 17.1 (Nov. 2018), pp. 1–36. DOI: 10.5381/jot.2018.17.1.a2. URL: https://hal.inria.fr/hal-01920362.

[42] TOPLAP. *About | TOPLAP*. Accessed 2019-02-22. URL: https://toplap.org/about/.

[43] D. Yellin and R. Strom. "INC: A Language for Incremental Computations." In: *SIGPLAN Not.* 23.7 (June 1988), pp. 115–124. ISSN: 0362-1340. DOI: 10.1145/960116.54002. URL: http://doi.acm.org/10.1145/960116.54002.

[44] E. G. Zmölnig J. "Live Coding: An Overview." In: *Proceedings of the International Computer Music Conference*. Copenhagen, Denmark, 2007.