HAL open science

Concepts and Tool for Interactive Computing

Alice Martin

To cite this version:

Alice Martin. Concepts and Tool for Interactive Computing. Human-Computer Interaction [cs.HC]. ISAE - Institut Supérieur de l'Aéronautique et de l'Espace, 2022. English. NNT : 2022ESAE0050 . tel-04030945

HAL Id: tel-04030945

https://enac.hal.science/tel-04030945

Submitted on 15 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par

Alice MARTIN

Le 24 novembre 2022

Concepts et outil pour l'informatique de l'interaction

Ecole doctorale : **AA - Aéronautique, Astronautique**

Spécialité : **Informatique**

Unité de recherche :
ENAC-LAB - Laboratoire de Recherche ENAC

Thèse dirigée par
Stéphane CONVERSY et Mathieu MAGNAUDET

Jury

M. Stéphane HUOT, Rapporteur
M. Michel BEAUDOUIN-LAFON, Rapporteur
M. Simone MARTINI, Examinateur
M. Stéphane CONVERSY, Directeur de thèse
M. Mathieu MAGNAUDET, Co-directeur de thèse
Mme Virginie WIELS, Présidente

Doctoral dissertation

Institution:
National French Institute of Aeronautics and Space

School/program:
AA - aeronautics, astronautics
Field: Computer science

Concepts and tools for interactive computing

PhD candidate : Alice Martin

Supervisors:
Stéphane Conversy, Mathieu Magnaudet
(ENAC)

Examination commitee:
Michel Beaudouin-Lafon (Paris-Saclay)
Stéphane HUOT (INRIA)
Simone Martini (Università di Bologna)
Virginie Wiels (ONERA)

This work was supported by a doctoral scholarship from Agence de l'Innovation de Défense (AID) of the Direction Générale de l'Armement (DGA) and by the French "Programme d'Investissements d'avenir" ANR-17-EURE-0005 conducted by ANR.

Acknowledgments

I want to thank my supervisors, Stéphane Conversy and Mathieu Magnaudet, for the risk they took regarding the proposed exciting topic and the trust they placed in a candidate who did not have the expected profile in computer science. It was the perfect opportunity to continue my training and to understand, in retrospect, better the questions studied years earlier on the notion of computation while getting the chance to be in contact with the aerospace domain. I am grateful to the patience of the members of the Interactive Computing lab at ENAC for helping me reach volunteers and conduct interviews and experiments.

I warmly thank my jury members, Michel Beaudouin-Lafon, Stéphane Huot, Simone Martini and Virginie Wiels, for their enthusiasm and benevolence that moved me and the good memory that will remain from the defense. I am also grateful to Didier Bazalgette, the DGA and ISAE-Supaéro for the funding provided and the doctoral school - particularly Maryse Herbillon and Catherine Mabru.

Some words exchanged by email or in person have helped me more than the interlocutors may have thought. I am thinking of reading advice and suggestions from Colin Klein, Edward Lee, Liesbeth De Mol, Marc Pouzet, and Nick Wiggershaus. Since the motivation for the thesis came mainly from the questions posed by philosophers and computer scientists in analytic philosophy, I am very grateful for the training I received at the Ecole Normale at Institut Jean Nicod and for the year-long visit that I had the chance to make to Rutgers University. In particular, I would like to thank Liz Camp, Carolina Flores, Michael Murez, and François Recanati. I am also grateful to Benjamin Icard, Pierre Trefouret, Frédéric Fogacci and Wendy Carrara for their thoughts and advice that helped me think about the next post-thesis adventures.

These three years have also been made of laughter with Pascal Béger, Nicolas Nalpon, Florine Simon and Vinitha Gadiraju. Last but not least, I owe my good thesis memories to my family and friends, to Agostino, Beate and Oliver.

Résumé

Les systèmes informatiques actuels au cœur des systèmes critiques, notamment le transport aérien, sont caractérisés par de multiples interactions, ou couplages forts, entre les opérateurs humains, les dispositifs physiques et les logiciels. La conception de ces systèmes nécessite de prêter attention aux relations causales entre les différents processus impliqués. Par conséquent, il ne s'agit plus de concevoir des systèmes d'entrée/sortie pour lesquels l'algorithme doit être créé, mais de spécifier des réseaux dynamiques de processus hétérogènes en interaction. En conséquence, ces systèmes informatiques ne peuvent plus être facilement appréhendés dans le cadre théorique classique : la théorie de la calculabilité, héritée des travaux de Turing et Church. Les événements asynchrones, les flux d'exécution indépendants, la création dynamique d'objets, ou encore les processus d'attente passive posent des difficultés spécifiques dans la modélisation et la pratique. L'objectif principal de cette thèse est d'examiner la possibilité d'un nouveau cadre théorique pour l'informatique interactive afin de mieux la caractériser, en suivant un programme de recherche qui vise à définir l'interaction. Sonder la question de l'interaction se situe à l'intersection entre l'interaction homme-machine et les pratiques de programmation impliquées, l'épistémologie de l'informatique et l'informatique théorique. Nous cherchons à expliquer ce qui rend possible l'interaction dans un système informatique, autrement dit nous nous interrogeons sur les mécanismes sous-jacents. Nous proposons le concept de modèle d'exécution pour construire une telle explication. Parmi les exigences, nous définissons la nécessité d'un composant que nous appelons un "orchestrateur causal". La conséquence de cette réflexion épistémologique est de motiver, guidée par une étude auprès de programmeurs, une approche visant à outiller un langage dédié à l'interaction. A travers un ensemble de techniques d'interaction au sein d'un éditeur de code, *Causette*, nous proposons d'aider le programmeur à comprendre les relations causales d'un programme.

Mots-clés: interaction, programmation, calculabilité, épistémologie de l'informatique, causalité, compréhension de code

Abstract

Current computing systems at the heart of critical systems, especially air transport, are characterized by multiple interactions, or strong couplings, between human operators, physical devices, and software. The design of such systems requires attention to the causal relationships between the different processes involved. Therefore, the task is no longer about designing input/output systems for which the algorithm must be created but specifying dynamic networks of interacting heterogeneous processes. As a result, these computing systems can no longer be easily understood within the classical theoretical framework: computability theory, inherited from the work of Turing and Church. Asynchronous events, independent execution flows, dynamic object creation, or passive waiting processes pose specific difficulties in modeling and practice. The primary objective of this thesis is to examine the possibility of a new theoretical framework for interactive computing to characterize it better, following a research program that aims at defining interaction. Probing the question about interaction is at the intersection of Human-Computer Interaction and involved programming practices, the epistemology of computing, and theoretical computer science. We are looking for an explanation of how interactive computing comes about and what mechanisms support it. We propose the concept of an execution model to build such an explanation. Among the requirements, we define the necessity of a component that we call a "causal orchestrator". The consequence of this reflection is to motivate, guided by a study with programmers, an approach to tool a language dedicated to interaction. Through a set of interaction techniques within a code editor, *Causette*, we propose a way to support the programmer in understanding causal relationships between processes described in interaction programs.

Keywords: interaction, programming, computability, epistemology of computing, causality, code understanding

Contents

Acknowledgments **1**

Résumé **3**

Abstract **5**

Introduction **13**

Context 13

"Interaction": a research program in computer science 13

Interaction: in what sense? 15

General definitions and scopes 15

Terminological clarifications 19

Restating the problem 22

The epistemological problem: a gap to fill to *explain* interactive computing 23

The practical problem: programming interaction, not only algorithms 27

Why does this research program matter, and what do we have to offer? 28

Strategy 29

Methodology 29

Dissertation overview 30

Contributions 32

Publications and talks 33

1 What is interaction programming? 35

1.1 Interviews with professional programmers 36

1.1.1 Participants 36

1.1.2 Method 39

1.1.3 Results 40

1.1.4 Analysis 42

1.1.5 Interview sum-up 46

1.2 A few dedicated models 47

1.2.1 The architecture of reactive *computational artefacts* — Suchman 48

1.2.2 Fundamental structure and concept of interaction code — Letondal and al. 49

1.2.3 The Anatomy of Interaction — Basman et al. 50

1.2.4 A system engineering model for interactive systems — ICOs notation 51

1.3 Dedicated frameworks and languages 52

1.4 Interaction programming challenges stated in the literature 54

1.4.1 Semantic and syntactic concerns 54

1.4.2 Execution concerns 56

1.4.3 Understanding concerns 58

1.5 Summary 59

2 Interactive computing in theoretical computer science 63

2.1 Milner: interactional vs. computational 65

2.1.1 Motivation 65

2.1.2 Account for interaction 66

2.1.3 Legacy 67

2.1.4 Issues for an account of interactive computing 67

2.2 Reactive TMs: extending the original model 68

2.2.1 Motivation 68

2.2.2 Account for interaction 69

2.2.3 Legacy 70

2.2.4 Issues for an account of interactive computing 71
2.3 Going beyond TMs? Wegner's new paradigm 72
2.3.1 Motivation 72
2.3.2 Account for interaction 75
2.3.3 Legacy 76
2.3.4 Issues for an account of interactive computing 77
2.4 Summary 78

3 Formal models of interactive computation vs. explanations 81
3.1 Equivalence and powerfulness of models 82
3.2 Formal models and mechanistic models 85
3.2.1 Digital computers, computing mechanisms and computability theory 87
3.2.2 What models of computation cannot explain: digging into details 91
3.2.3 Execution models vs. models of computation 94
3.2.4 Positing the concept of execution model among kinds of computational explanations 98
3.3 Summary 102

4 Conceptual proposal: an execution model for interaction 105
4.1 The execution model for interaction 106
4.1.1 Specifics of an interactive execution model 106
4.1.2 Minimal requirements 107
4.1.3 Components 112
4.1.4 Refining the causal orchestrator with dynamicity concerns . 114
4.1.5 Mechanistic description 115
4.2 Linking the execution model with existing interaction languages . . 118
4.2.1 From the execution model to interaction semantics 119
4.2.2 Interaction expressiveness of existing languages 120
4.3 Summary 123

5 Practical proposal: *Causette*, interaction techniques to support causality understanding 125

5.1 Literature survey to support the design of Causette 126

5.1.1 How causality has been addressed in interactive programming 126

5.1.2 Augmenting textual code in IDEs 126

5.1.3 Code representation and animation 127

5.2 Informal ideation with our 12 interviewees 128

5.3 Requirements and Design principles 129

5.3.1 Requirements 130

5.3.2 Design principles 130

5.4 Interactions 131

5.4.1 Interaction 1: reordering a data-flow 131

5.4.2 Interaction 2: reordering *textual* FSMs 134

5.4.3 Interaction 3: reordering *graphical* FSMs 136

5.4.4 Interaction 4: showing the dynamics of FSMs 137

5.5 The Smala language and Causette's implementation 138

5.6 Evaluation 139

5.6.1 Research questions 139

5.6.2 Participants 139

5.6.3 Experimental design 140

5.6.4 Results 143

5.7 Threats to validity 151

5.8 Summary 153

Conclusion 155

Contributions 155

Implications 157

Limitations 158

Future work 159

Broader scope and motivation 159

Bibliography 161

A Smala's syntax **189**

B Interview transcripts **191**

Introduction

Context

"Interaction": a research program in computer science

In the 1990s, Wegner presented radical reflections opposing classical algorithmic computing to interactive computing [285, 286, 287, 288, 289]. His paper from 1997 has remained famous through the following slogan: "Interaction is more powerful than algorithms" [286]. The claim was that a new framework or even a new computing paradigm was required to account for contemporary computing. It was argued that the feature of interest in contemporary computing was the ability of systems to react to diverse processes of the external environment.

Wegner's reflections on the possibility of a new paradigm were inherited from questions already posed by Milner as early as 1975 [179, 180, 181, 182]. Milner had discussed the concept of interaction in computer science and had introduced a distinction between *computational* and *interactive* behavior. Wegner pushed the distinction further by hypothesizing a new paradigm. A research program on interaction has emerged since. Other slogans have followed, like "Interaction, the Future of Programming" proposed by Bret Victor in a conference in 2013 entitled "The Future of Programming" or "Interaction is the Future of Computing" by Michel Beaudouin-Lafon [22].

One reason for the research program to remain active is that today, every usable computing device, from the smartwatch on our wrist to the complex computing systems embedded in the cockpit of an aircraft, relies on a complex entanglement of interactions between incoming external events and computational processes. To Wegner, but also to many proponents in various computing fields, these devices raise new challenges. Contemporary devices exhibit some properties that were not

originally expressed within the classical theoretical framework associated with computing: computability theory and its extensions, used as cornerstones in the mathematical view of computing. We can point, for example, to the significant emergence of a new programming paradigm labeled "reactive programming", which, according to its practitioners calls for new concepts [16, 35, 114, 245, 250]. Another example could be the observation formulated by the Human-Computer Interaction (HCI) community: interactive software requires the verification of new properties, which are not, strictly speaking, classical properties of computation, e.g., graphical properties [48, 86, 213]. Therefore, Wegner's question is still at stake: to what extent do current computing systems need a new conceptual framework? This reflection can be found in several communities that each, with their own perspective, question the theoretical foundations of computer science: the field of HCI [21, 22, 57, 153], the cyber-physical systems community [143, 145, 146], the epistemology of computing [1, 2, 82, 162] in particular.

It is worth reminding that computing is a matter of science and technology, logic and engineering [284], and both dimensions do not evolve at the same pace [182, 195, 196]. Sometimes scientific theories and models are ahead of practices and are referred to as models for practices in an ad-hoc manner. In that respect, computability theory predated the invention of the first physical computer. It is debatable whether computability theory actually inspired practices [75, 76] and described practices adequately. However, computability theory retrospectively became a mathematical framework to think of computing as a mathematical activity [173]. Sometimes practices might be ahead: computers did not wait for theories of communicating processes to run and have threads communicate. In any case, we can probably say that the history of computing is made of back and forth between adjustment of theory to practice or practice to theory. Adjusting in one direction rather than another characterizes different programs or agendas in computer science [173].

Wegner's proposal is on the side of adjustment of theory to practice. Our work aligns with this research program about interaction: reflecting on concepts and models dedicated to interactive computing systems. It does not pretend to settle the debate but proposes a conceptual "map of the territory" and a strategy to tackle the question. The thesis also has a concrete objective: to develop a practical approach to help programmers program interactive systems and understand what we call "interaction code".

Interaction: in what sense?

General definitions and scopes

The notion of interaction since Milner and Wegner has multiple uses across computing-related disciplines, from theoretical computer science to Human-Computer Interaction (HCI) and the philosophy of computing. Therefore, it is worth starting with some definitional attempts and then going through the different uses of the concept to clarify our work. As a first approach, let us have a look at its generic definition, such as found in a dictionary: interaction is defined as "reciprocal action, or coupling" [1] (Larousse). In that broad sense, interaction refers to phenomena with a mutual influence between two bodies, e.g., the gravitational force. When it comes to interactive computing systems, there are degrees of coupling. The strongest kind of coupling is when mutual influence is at stake. This is the case for some haptic devices where a user interacts with a computing system constraining the user's gesture (e.g., the Phantom device [2]). In that strongest sense, the coupling can be bidirectional. In other more common cases, the notion of interaction applied to computing systems is more unidirectional. For example, when a user inks a sketch in a drawing application (coupling from the user to the computing system) or when the computing system sends a haptic notification to the user (coupling from the computing system to the user). In any case, some concept of *causal relationships* is warranted to describe the couplings in whichever form they come. What is at stake in interaction is the orchestration between some action triggering certain effects in an execution and feedback loop. That orchestration is required to fit the system designer's intentions.

We think our thesis belongs to a certain view on interaction recently expressed in "What Is Interaction" presented at CHI in 2017 by Hornbaek and Oulasvirta [120] and hope our thesis can provide some propositions in that direction:

> "The term interaction is field-defining, yet surprisingly confused. This essay discusses what interaction is. We first argue that only few attempts to directly define interaction exist. Nevertheless, we extract from the literature distinct and highly developed concepts, for instance, viewing interaction as dialogue, transmission, optimal behavior, embodiment, and tool use. Importantly, these concepts are associated with different scopes and ways of construing the **causal relationships** between the human and the computer. This affects their ability to inform empirical studies and design. Based on this discussion, we list desiderata for future work on interaction, emphasizing the need to improve

[1] https://www.larousse.fr/dictionnaires/francais/interaction
[2] https://fr.3dsystems.com/haptics-devices/3d-systems-phantom-premium

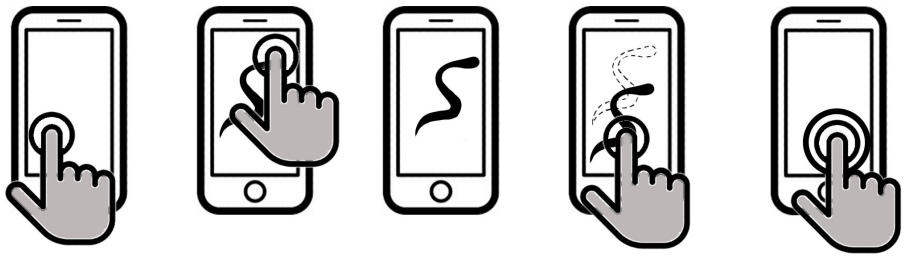

Figure 1: Example of an interactive system: a drawing app

> scope and specificity, to better **account for the effects and agency** that computers have in interaction, and to generate strong propositions about interaction." (The bold characters are ours.)

To overcome the confusion mentioned in the previous quote, let us first explain what *interactive computing systems* refer to and in what context the notion is used. Historically, computing systems were first conceived and thought of as performing mere computations, being closed and *transformational* systems in charge of taking as input a series of instructions to be transformed step-by-step, without interruption, until the production of an output result [114, 151, 152, 168, 182, 183, 184].

To understand what makes the specificity of interactive computing systems as opposed to transformational ones, consider this simple example: a drawing application on a smartphone. That does not mean that this example covers every interactive scenario of interest and every degree of possible couplings. However, we chose it for the sake of simplicity and because that kind of interaction has become ubiquitous [3]. When one touches the screen and moves a finger over it, the application draws a line whose thickness depends on the pressure applied to the screen. Once the pressure is released, the drawing becomes available as an object to be interacted with. If the user does a double tap over the screen, the drawing is erased as in Figure 1.

As simple as it is, this example reveals some interesting phenomena:

- A physical event triggers the drawing. Thus, there is a causal link between physical and computational processes.
- One property of the drawing (the thickness of the line) is entirely dependent

[3] Following some estimates, there are 6,6 billion smartphone users worldwide. See `https://www.bankmycell.com/blog/how-many-phones-are-in-the-world`

on the pressure of the finger, making the computational process responsive to the structure of the physical event.

- During the execution, the drawing becomes a new object on the user interface and can be interacted with. In other words, the production of outputs is dynamic (there is no need to wait for the execution's termination), and outputs can themselves become new inputs for subsequent actions.
- The double-tap behavior involves measuring the time that has elapsed between two taps.

Unlike a transformational computing system, an interactive computing system allows reactions between physical phenomena in time. There is no more a finite execution signified by the result of some calculation. Instead, an interactive system relies on an execution loop without a final result. In other words, it is a system that supervises in time a predefined number of possible couplings between processes, including processes external to the system. A mere transformational system, once the execution of a pre-determined calculation has been launched, cannot react to any action.

Even if, historically, the notion of computation has been used as a theoretical framework for computer science, computing systems themselves *in practice* have remained purely transformational for a short time only. They have very quickly presented properties or modes of interaction between user and machine, giving them an essential interactive dimension. We can mention, for example, the need to master and formalize machines running processes in parallel as early as the 1960s, with the growing challenge of having them communicate and synchronize. As for the modes of interaction between user and machine, one can think of the Read-Evaluation-Print loop (REPL) environments that accompanied functional and imperative programming already in the 1960s [4]. Thus, the interactive dimension of computing systems is far from new and has evolved with increasingly rich properties coming to the fore, from communication between competing processes to the possibility of multimodal interactions between humans and systems that we currently know. Any useful computing system reacts to various input events (e.g., server request/response, peripheral inputs from the keyboard, mouse, touch screen). Sometimes, interactive systems are materialized for users by graphical user interfaces, which constitute familiar examples. Still, as underlined by researchers in the HCI community [20], interaction is a broad phenomenon that cannot be reduced to interfaces.

[4]For the implementation of the LISP language at the origin, see [32, 79]). A read–eval–print loop (REPL) is an interactive computer programming environment. It takes single user inputs, executes them, and returns the result to the user. The term usually refers to programming interfaces similar to the classical Lisp machine interactive environment.

Today, when we talk about interaction in computer science, the territory is vast, and the notion of interaction is theorized within different sub-disciplines, among which at least:

- Theoretical computer science: the discipline asks whether systems that are not purely transformational require new models and formalisms. A distinction between classical and interactive computation is offered [182, 183, 286, 292].
- Philosophy of computing: from that perspective, the question is whether the epistemology of computing needs to be updated [82, 162]. This discipline is close to the previous one as they have epistemological questions in common.
- Software engineering, targeting the programming of interactive systems: the research program aims at facilitating the programming of interactive computing systems at the level of software architecture and language structures [10, 16, 28, 44, 53, 57, 58, 122, 126, 153, 164, 167, 200, 203, 208, 210, 213, 218, 252]. Indeed, there are specific problems related to these systems. The general problem is managing an interdependent behavior of many components and the uncontrolled and unpredictable flow of external events.
- Interaction design is not understood as programming. However, it focuses on the interaction modalities between users and systems: how to model and enrich the communication between the human and the system [20].

The territory is all the more vast as an interactive system can be analyzed at different levels. We can refer to the stratification proposed by the INRIA Loki team [5]:

- "micro-dynamics": "low-level problems related to interaction such as studying transfer functions, latency compensation, and tactile feedback."
- "meso-dynamics": "augmenting the interaction bandwidth and vocabulary."
- "macro-dynamics": "real-time activity monitors and better system adaptability for skills acquisition while using those systems."

Before going further into the landscape analysis, let us specify from the outset which disciplines we will elaborate on and at which level of analysis. As far as the

[5] https://loki.lille.inria.fr/

disciplines are concerned, we will deal with the first three to both deal with epistemological questions about interactive systems and tackle practically the difficulties linked to the programming of these systems. Our level of analysis is at the micro level and even lower if we refer to the scale above. We leave aside user-centered concerns (such as "touch feedback" or "latency compensation") to focus on an intermediate level of abstraction between the physical implementation of the system and the high-level language coding the system.

Terminological clarifications

We focus on interaction as referring to the set of systems that, contrary to closed computing systems carrying out a classical algorithmic computation, can interact with an environment and, in particular, with a human agent. To avoid confusion about the concept of interaction that we will use in this thesis, we prefer to take the time to clarify the various existing meanings and specify the spectrum of our concept.

Interaction as defined in the field of Human-Computer Interaction presents a spectrum of interpretations, from human-centered [20, 23] to machine-centered account [213]. In HCI, interaction can refer to (i) how humans transmit information to machines; (ii) how machines transmit information to humans; and (iii) how the link between reception and emission of information between humans and machines is implemented in code [235]. When we talk about *interaction*, we can therefore focus on the machine's peripherals to enrich the interaction and its modalities, for example with the use of mobile devices, finger and body interaction, or immersive navigation in 3D virtual worlds. A possible perspective with these two aspects in mind is to dig into cognitive and sociological aspects of human behavior interacting with computing systems. We chose to leave these aspects aside.

This thesis is interested in the software and hardware dimensions of interaction. We want to understand how interaction comes to be, in other words how the link between users and machines is possible. We will focus on *interaction* as referring to *interactive computing systems* and the programs that support them. We are interested in their modeling (theoretical aspect of the thesis) and their programming (practical aspect). Thus, when we look for a model for interactive computing systems, we will not be looking at the existing socio-technical theories of interaction but at models and theories available in theoretical computer science, philosophy of computing, and software engineering. On the practical side, we are interested in the programmers of these systems.

Let us, therefore, define what we mean by *programming interaction*.

Programming interaction encompasses the programming of *interfaces* and the programming of *interactions* (notably *interaction techniques*). Both are translated into code and can therefore be considered programs.

- *Interaction techniques* are ways of interacting between humans and machines, for example, to click on a drop-down menu to display a list of options. Tucker [274], and Foley [93] have provided classical definitions: "An interaction technique is the fusion of input and output, consisting of all software and hardware elements, that provides a way for the user to accomplish a task" [274] or "a way of using a physical input/output device to perform a generic task in a human-computer dialogue" [93]. Interaction techniques are ideally consistent, i.e., users can anticipate how they will work before interacting and understand what is happening during the interaction. For example, if one clicks on a drop-down menu, one expects a list of options to be displayed and the visually highlighted option to be selected. Because they are usually expressed in programs, interaction techniques are very representative of interaction programming: they are programs that interact with users.

- The *interface* materializes the object of interconnection between human and machine. By contrast, interaction refers to the whole phenomenon of information exchange between humans and machines or between various machine processes. When programming interaction techniques, programming always results in concrete code and will necessarily be materialized by an interface. When it is visible on a screen, it is called a Graphical User Interface.

In the literature, several work are related to *interactive computing systems* and *interaction programming*:

1. *Reactive systems*:

   A classical definition can be found in Boussinot's work:

   > "Reactive systems have been defined by Harel and Pnueli as systems that are supposed to maintain an ongoing relationship with their environment. Such systems do not lend themselves naturally to the description of functions and transformations: they cannot be described adequately as computing a function from an initial state to a terminal state. On the contrary, behaviors of reactive systems are better seen as reactions to external stimuli. The role of reactive systems is to react continuously to external inputs by producing outputs. For example, man-machine interface handlers or computer games fall into the category of reactive systems." [37].

In addition to that definition, a further distinction is introduced in the reactive programming community: reactive systems are sometimes distinguished from interactive systems [114, 168]. A distinction is made around the real-time dimension of these systems. An *interactive system* reacts to events in the environment without time constraints, whereas a *reactive system* reacts within a time limit set by the environment. For example, the kernel of a general-purpose operating system (OS) is interactive (its response time to events depends on its load and hardware capabilities). By contrast, the autopilot of an aircraft is a reactive system (its response time to events is specified and must be respected).

2. *Reactive programming*:

   Similarly to "reactive systems", the phrase "reactive programming" can raise some ambiguities because it can be associated with two different styles of programming with different conceptions of time, either logical time (one refers then more precisely to "synchronous reactive programming") or absolute time ("reactive programming" in a loose sense). In some papers, the distinction is not stated. Still, some authors remind that "reactive programming" embraces two styles of programming:

   > "Recently, several languages have been designed for reactive programming. Among these reactive languages, we can cite the imperative language Esterel, two dataflow languages, Lustre and Signal, and the graphical specification formalism Statecharts. These languages do not use an absolute time as is the case, for example, in Ada with the delay statement. Instead, they use a logical time divided into instants, which are moments when programs react. In Esterel, we would write await 3 S to wait for the third instant where the signal S is present (no matter what the signal S denotes). This approach leads to a new programming style where one programs in terms of reactions to activations and one thinks in a logic of instants." [37].

   On the contrary, as can be found in Bainomugisha [16], "reactive programming" can be taken to refer directly, without any further distinction, to non-synchronous programming:

   > "Reactive programming has recently gained popularity as a well-suited paradigm for developing event-driven and interactive applications. It facilitates the development of such applications by providing abstractions to express time-varying values and automatically managing dependencies between such values. Several

approaches have been recently proposed embedded in various languages such as Haskell, Scheme, JavaScript, Java, NET".

What we call *interaction programming* refers to the programming of interactive systems involving a human user in a broad sense. This has the advantage of avoiding the term "reactive programming", which is more common but used in two different ways: sometimes within the framework of the synchronous hypothesis [25, 28, 53] and sometimes without reference to temporal constraints [16, 245].

3. *Interactive computing*

   In theoretical computer science, reflections on models for new computing practices introduced the term "interactive computing" to embrace a wide set of properties that allows internal communication between machine processes and the possibility of interacting with external drivers [105, 267, 286]. It is also a notion that has come to the stage in the philosophy of computing [82, 162]. The perspective, in that case, is focused on the compatibility of interactive computing systems with classical models of computation.

Therefore, we use the term "interactive" and "interaction" in a broad sense, encompassing its uses from the HCI community, from epistemologists (philosophers of computing and computer scientists), and the *reactive* and interactive programming communities. We prefer a looser use of the term and group the previously distinguished notions under a single encompassing category: interactive computer systems with human users involved, requiring a general model.

As much as functional programming is about programming functions that run within a program that computes results, *interaction programming* is about programming interactions that run within an interactive program, i.e., that reacts to external events such as human input or network data reception. *Interaction code* is the code devoted to describing interactive programs.

Restating the problem

As we have previously stated, there is hardly a computing system today whose functions and usefulness are not based on its ability to react satisfactorily to events. These systems present specific challenges on two levels that we want to consider in this thesis: both an *epistemological* challenge (how we can describe, understand, and model these systems) and a *practical* challenge (how to facilitate their design). On the **practical level**, the difficulty stems from the fact that programming an

interactive system does not require solving algorithmic tasks only. On the **epistemological level**, the difficulty is to define adequate conceptual tools to describe interactive systems.

The epistemological problem: a gap to fill to *explain* interactive computing

On the epistemological side, the problem is that we do not have enough conceptual tools to account for the specifics of interaction programming practices and their challenges within a general theory. Furthermore, we do not know what form a general theory should take. We mention in the following two precise issues for the epistemology of computing regarding interaction. We show that the epistemological framing of questions about computing is made in terms of classical problems and models from computability theory. We underline that such a framing is astonishing since computer science and computing have had since the early days of computing other frameworks than classical computability theory [175]. However, this framing is worth observing and being explained, as it is sometimes present outside the epistemology of computing and is still present in programming views. For example, through the concepts of *Turing completeness*, the expressiveness of a language depends on the computable functions it allows to express [38]. Such a view involves an idealization commented on recently by Martini [173].

The first issue is due to the early framing of the debate about interaction in terms of implications for the Church-Turing thesis [64, 233]. Wegner's slogan ("Interaction is more powerful than algorithm") suggests that "interaction" is a kind of hypercomputation, computing more than the set of computable functions. Wegner's commentators or objectors tend to worry that a theory of interaction carries a threat against the Church-Turing thesis. Therefore, they discuss extensions of the Turing Machine, e.g., with oracles [6]. The question is whether a classical model

[6]Turing introduced "oracles" in his Ph.D. dissertation [276]. Turing had extended the automatic machine (what is usually referred to as the "Turing Machine", as described in Turing's seminal paper [275]). Turing had thought about formalizing the solving of uncomputable problems. In an automatic machine, all data is given before the execution starts, and there is no means to change the symbols on the tape once the execution is launched (the tape header can write and erase symbols — but these are given prior to execution). But an oracle machine can consult an oracle during an execution step, being provided with a new symbol (possibly an uncomputable one) during execution. The halting problem becomes then solvable. Turing's work was expanded later by Post [231]. See, e.g., Soare's work [267, 268] for a more detailed study on the introduction of oracles by Turing and how Post expanded Turing's ideas. Turing's and Post's work has later inspired derived formalism on "extended" Turing Machines, such as the "Reactive Turing Machine" [9, 15, 150, 159] or the "Persistent Turing Machines" [104], that we detailed in Chapter 2.

of computation with some extensions can express and formalize the properties of an interactive system. Therefore, as "interaction" arrived on stage as a concept needing a theory, many answers were framed with references to a very specific view of computing: not only a mathematical view but one shaped by classical computability theory.

We argue that references to the Turing Machine and the Church-Turing thesis are not the only adequate concept and theory to reflect on interaction. We argue that this framing is not necessary and that *models of computation* have not been sufficiently distinguished from models of *computing systems*. That could explain why proposals on interaction claiming that "interactive systems do *things* that classical models of computations cannot express" **(claim 1)**, are immediately interpreted as "computing systems compute *more functions* than what models of computation can express" **(claim 2)**. But this does not need to follow once "computing" (what computers do) is distinguished from "computation" (an effective procedure corresponding to specific formalisms). Let us take a mundane example to make this more intuitive. Let us say that Alan is a good cook but cannot swim; let us say that Emil can swim AND cook. But it does not imply that Emil cooks better than Alan. In the same way, there is no reason to deduce claim 2 about computing from claim 1. The lack of distinctions between *computation* and *computing* is made even more salient in the vocabulary we use: there is no word to refer to *computers* or *computing systems* deprived of reference to computation. The premise of our work is that the mentioned distinctions should be taken seriously. It guides the search for a dedicated account for interactive computing systems to analyze and support interaction programming practices.

The second issue concerns a persistent reference, particularly in the philosophy of computing, to the Turing machine as the adequate abstraction to understand not only computation but also physical computing systems. This has been commented on in the literature [162, 196, 291]. The epistemology of computer science and computing, particularly in analytic philosophy, still describes a computer system in reference to the Turing machine, its architecture, and its mechanisms at play [96, 225]. Among other models of computation, the automatic Turing Machine (or *a-machine* but we will use TM from now on [7]) presented by Turing [275] had an initial and narrow scope, that of the Church-Turing thesis [72]. The TM helped define what is computable by effective means, and the TM is proven to be equivalent to other formalisms within computability theory (such as the lambda-calculus [62]). The TM became philosophically influential and later, within and

[7]Since Turing himself in his thesis introduced several theoretical, abstract machines, e.g., the *o-machine* or oracle machine, it can be convenient to use Turing's label for the abstract machine he presented in the 1937 paper [275] and which is the standard reference philosophers have in mind when talking about the "Turing Machine".

outside computer science, a reference to think of what computers do [196, 266]. This broader scope assumes that computing systems can sufficiently be understood through classical models of computation and that programs can be understood as describing algorithmic systems.

Therefore, as we said from the outset, the epistemology of computing still presents a specific lens, namely that of computability theory. On the one hand, the initial project for computability theory was to answer a mathematical problem. On the other hand, the theory preceded the invention of the first physical computer. Computability theory is the result of pioneering work by logicians to answer the problems posed by Hilbert's program [211]: the possibility of formalizing any mathematical reasoning completely. In that respect, the Turing machine proposed by Turing in his 1937 paper [275] aimed at formulating rigorous proof concerning the decision problem for first-order logic. Turing's proof showed that no effective procedure could decide first-order logical provability. The strength of the Turing paper, and some would say its superiority over other equivalent formalisms [8] [259], is to have been able to define this intuitive notion of an effective procedure or mechanical procedure or algorithm used by mathematicians.

> "To the question "What is a 'mechanical process?" Turing returned the characteristic answer 'Something that can be done by a machine,' and embarked in the highly congenial task of analyzing the general notion of a computing machine (ibid.)." [214]

Such a process referred to as a *computation*, consists of sequences of operations on symbols carried out by a mathematician, or any human "computer" with a pencil and a piece of paper, or by mechanical devices without any thinking, intuition, or guess. Operations were carried out according to a finite number of rules. This proof in logic has given rise to anachronistic interpretations. One of them concerns us particularly in this work: the idea that the concept of computation defined through the kind of effective procedure described by Turing refers to everything that a computing machine can do. Because of the mechanistic intuition it conveys, it is historically understandable but striking that it became widely assumed in an ad-hoc manner that the TM was saying something about computers that did not even exist in 1937. The philosophical influence of the TM within and outside computer science to think of what computers do has been commented on in the literature [196, 266]. By extension, the TM also became a model for thinking of information processes in brains [123, 188, 261].

We want to argue that the value of the TM for epistemologists has relied on the intuition the TM provides on an execution mechanism. In other words, the TM

[8] The strength of Turing's formalism compared to Church's was commented on by Gödel. See Shagrir's synthesis on that issue [259].

could provide a model of computation. But it could also, contrary to equivalent formalisms like the lambda-calculus, describe (although in a very abstract fashion) *how* the execution of that computation is carried out. Our project defends the idea that what the Turing Machine as a model could do for *computation* has no equivalent for *interaction*. What made the Turing Machine stand out was its ability to provide both a formalism for computation and the intuition of how the computation could be carried out on a physical device.

We think that, when describing interaction, we lack that same level of abstraction available in the TM: a general execution model, providing a hint of how execution is carried out, with some references (although abstract) to the implementation mechanisms at stake.

The objective of the thesis is, therefore, the following. We will show that we are left with a theoretical gap to fill: we need a general and minimal model for interactive computing systems like the Turing machine was for classical non-interactive computing. That does not mean that current interactive computing machines do not compute; but they also do many other things requiring a new general model. To go back to our example illustrated in Figure 1, the relevant phenomena at stake making interaction possible between the user and the application are not reducible to calculations. We need, e.g., to explain how the drawing can become an object dynamically during the execution or how there can be a difference between a click and a double click. This requires mechanisms absent from the TM. By pointing at the limits of computability theory to account for interaction, we are not in any case challenging Turing and Church's seminal work. There is a priori no way the Church-Turing thesis could be challenged because there is likely no other alternative to formalize an algorithm or effective procedure. We intend to argue that computability theory did not account for interactive systems reacting to events, like human-computer interaction systems.

A model for interaction should be general enough to provide the intuition of how interactive computing comes about (how it is orchestrated and can be implemented). Supporting such an intuition would help in several contexts: introduction classes to the epistemology of computing and interactive programming and interaction design projects. In other words, it could serve as a grounding intuition like what the Turing Machine does to introduce classical computing and computation-oriented programming.

The practical problem: programming interaction, not only algorithms

The computability paradigm has influenced and framed programming practices. It might not leave enough space to facilitate the writing of new kinds of programs, where the description of algorithmic procedures is only one aspect of the programming activity. The problem has already been pointed out in the literature:

> "We suggest that the traditional view of programming is biased. Turing and the generations that came after him have created such a consistent body of theories and programming languages that the theory of computation is used ubiquitously for analyzing systems, designing algorithms, and even as natural science. This success sometimes obscures the existence (even the prevalence!) of other kinds of programs." [58]

The legacy of computability theory has resulted in programming practice being dominated by procedural and functional models. The function call is thus a classical pattern. However, it is a limiting pattern for efficiently [204] programming rich interactions [58].

We will see in this thesis that one way to capture the problems encountered by interaction programmers is that the core of their task is to program *causal relationships* between heterogeneous physical processes. Following the philosopher of science W. Salmon [244], we take *causality* as a spatio-temporal process involving the transmission of "information, structure and causal influence". We consider that two processes A and B have a *causal relationship* if A always precedes B and B occurs every time A occurs. In other words, "causality" means that a piece of code's execution follows an input event's occurrence. *Causal chains* are a set of elementary causal relationships that form a chain. Understanding causal chains adds to interaction programming supplementary tasks different from thinking only about the programming of procedures and functions. We aim to suggest principles for a debugging/understanding approach in line with previous work in program debugging.

We will present *Causette*, a set of interaction techniques to enhance a code editor. The set of interaction techniques targets the understanding of two code constructs involved, respectively, in dataflow and control flow: bindings and Finite State Machines (FSMs). We argue that they are examples of "causal programming constructs": syntactic expressions that establish a particular causality between two pieces of code.

Our main design principle is to bring together the causal relationships far from each other in the source code. We think the view of interaction programming that these interaction techniques support is valuable: a causal approach.

Why does this research program matter, and what do we have to offer?

The program stays relevant because of the important need to go back and forth between science and technology, theory and reality, which shapes what computing is. If the scientific theory is behind practices, then the theoretical account is outdated, and the mapping between the model and reality fails. This is an epistemological pitfall. It can have several unwarranted consequences, for example, in the view that programmers have of their programmed system or that students are taught in class. The usability of programming is at stake: if practices are behind the theory, the resulting situation may also be unwarranted, making the practitioners' activity harder, juggling with inappropriate conceptual tools. Programming a system with a language whose semantics is not fitting the tasks at stake might be time-consuming.

Although launched years ago, we think the research program is still worth pursuing, and the reason goes as follows. On the one hand, there is a general agreement on the opposition between calculators working like closed systems with all their inputs given before execution and computing systems that react in real-time to events. The distinction can be found with different labels, a famous one being the opposition between *reactive* and *transformational* systems, according to Harel and Pnueli [114]. On the other hand, we argue that no general dedicated abstraction is available.

What do we have to offer within that interaction research program? First, we propose to insert the question of the research program within the programmers' activity. Second, we want to identify possible barriers to an explicit and general theory of interaction. We argue that historically the debate around Wegner's work has approached interaction through a specific lens, reducing the debate to one question: is interaction reducible to a *Turing Machine*, and if not, is it a threat to the Church-Turing thesis? Interestingly, the few explicit theoretical models of interactive systems are often presented as extensions of the Turing Machine [14, 15, 267, 287, 293]. We argue that this focus and reduction of the question is problematic because it cannot explain the relation between interaction programs and their implementation. It also leaves aside many aspects that matter in programming practice.

Our interest in this research program is not only an isolated theoretical exercise. Reflecting on what ingredients are needed for a theory of interaction has practical consequences on what can be done with that theory. It notably has consequences on how to think of semantics to describe an interactive system and, therefore, can help circumscribe the needs in practice regarding dedicated frameworks, languages, or tools.

Strategy

Our strategy is to restrict the scope of the investigation. We will investigate interaction by looking at the reality of interaction programming practices. This choice is motivated by an empirical approach. In other words, we redefine the question of *what interaction is* (as opposed, probably, to computation) into *how interaction is programmable*. That means we will pay attention to programming practices and engage in a study of programmers' activities. That is why our practical aim is a proposal addressed to programmers: support in code understanding.

Our interest in programming also means that we are interested in the nature of interaction programs: *what do these programs need to express* and *how can they be executed*. Questions about the semantics and implementation of programs have mainly been addressed by the epistemology of computing, both by computer scientists and philosophers [130, 187, 227, 242, 254, 260, 266, 272, 294]. But until now, no extensive attention has been paid to interactive systems, and only a few exceptions can be identified [82, 162]. Epistemology of computing is a recent field [45, 194, 195, 196, 272]. The field motivates theoretical proposals to better explain computing models, concepts, and practices within the field, especially regarding interactive systems that have become ubiquitous. Therefore, the other tenet of our strategy is to formulate a theoretical contribution at a specific level of discourse: the epistemology of computing.

In so doing, we both have delimited the field of exploration and identified a frame of discourse.

Methodology

We combined several approaches to tackle our theoretical problem and then derived solutions to guide a tool. First, we wanted to gain insights from interaction programming practices. We organized interviews with professional interaction programmers. We thought their activity should exhibit specifics related to interaction. We also hypothesized that programming interaction could be conceptually understood as programming causal relationships. We conducted interviews to explore this hypothesis, to collect striking scenarios, examples of recurrent programming difficulties, and insights into the mental models elaborated by programmers to understand their system, implement and correct their code.

To complete this practice overview, we surveyed fundamental concepts proposed by the interactive programming community and the known challenges addressed in the literature. At the same time, we also looked at how interactive computing has been addressed in theory. That involves examining how theoretical computer

science has explicitly posited interactive computing in the computability theory frame.

This double journey led to a point where it is possible to compare theory and practice and evaluate the mismatch. The point of view of practitioners turned out helpful in overcoming the mismatch and guiding a proposal. We used methods and concepts from the philosophy of computing to conceptualize the reasons for the mismatch and help form new requirements for an account of interaction. The theoretical work ended up motivating a programming tool dedicated to interaction code understanding by circumscribing its conceptual concerns. We then used standard iterative HCI methods to develop the interaction techniques involved in the tool.

Dissertation overview

This thesis is organized into five chapters and proceeds as follows.

Chapter 1 offers a first look at practice. This chapter presents the results of 12 interviews with professional programmers of interactive systems (including web programming, GUI programming, video, etc.). We then review the landscape of available interaction frameworks and languages and synthesize the well-known challenges of interactive programming as presented in the literature. We complete this overview by mentioning the proposals or suggestions that favor dedicated concepts or general models for interaction within the interactive programming community.

Chapter 2 consists of a literature review in theoretical computer science. The gap between the theoretical foundations of computing and the reality of computing practice has already been commented upon by computer scientists. We identify the relevant fields in theoretical computer science that have explicitly addressed the problem and explicitly proposed a theory of interactive computing. By looking at interaction within its understanding in theoretical computer science, we want to know, in the first place, how computer scientists have conceptualized and formalized interaction and how they have compared it with classical models of computation - par excellence, the universal Turing machine. We will see that the first explicit formulation of the opposition between classical computing and interactive computing was given by Milner in his reflection on the synchronization between communicating processes. We will then examine existing work and proposals on the "Reactive Turing machine", which, to address the evolution of computing practices and the need for new models, takes up a formalism already introduced by Turing, that of the oracle machines; and finally, the interactive paradigm introduced by Wegner. This theoretical journey aims to identify which dimensions of interactive systems these theories precisely address or, in other words, to identify what interaction

theory means for each of them. This will allow us to examine the possible limits or criticisms they have received. We will also consider how they respond to our specific concerns on the side of interactive programming: can these theories support our understanding of interactive systems and account for the practical difficulties of programming them?

A result of Chapter 2 is that discussions have, above all, debated the reducibility of formal interactive models of computation to classical models. Ambiguities about the use of *expressive power* and *equivalence* between models have led objectors to interpret the interactive paradigm as an illegitimate threat to the Church-Turing thesis. Such worries, we argue, may rely on a narrow interpretation. In any case, the debate prevents theorists from characterizing an interactive system: to focus on the reducibility of an interactive model to a classical model is not an answer to our epistemological question.

Chapter 3 proposes a reflexive approach to the issues found in the literature review. To clarify the epistemological problems around the question of modeling interactive systems, we need to clarify the concepts of models and explanations and be clear about our explananda[9]. We will divide this reflection into three steps. First, we return to the notions of equivalence and expressive power of a model to make sense of the unanswered debate left in Chapter 2. On the one hand, we want to show that the formal equivalence between two models leaves the possibility that something has been lost in reducing one model to the other. On the other hand, we want to insist on the ambiguous use of the term "power" of a model, which can refer specifically to computational power or expressive power more broadly. Then, we will propose to delineate more precisely why the debate identified at the end of Chapter 2 does not seem to provide a satisfactory epistemological answer. It cannot explain the phenomena on which the very possibility of an interactive computing system rests. We will introduce a distinction between *formal* and *explanatory mechanistic* models. We will deduce that formal models for interaction (such as the explicit models surveyed) and, more broadly, formal models allowing verification of properties of interactive systems cannot explain how a computing system can be interactive. Finally, we will conceptualize the nature of the abstraction that can serve such an explanation: an *execution model*. In our terms, an execution model is an intermediate abstraction between the program and its physical implementation, which allows describing in mechanistic terms how the execution of a program is realized by identifying the components of the execution mechanism. This implies proposing for interaction what the Turing machine proposed for the execution of a computation: identifying a minimal functional architecture and relationships between its components.

As announced in our introductory remarks, our objective is twofold and should

[9] *Explananda* = what needs to be explained.

lead to two proposals: theoretical and practical. The first theoretical proposal is formulated in **Chapter 4**. We propose to define an execution model for interaction. To evaluate the relevance of the proposed model, we list languages and frameworks dedicated to interaction and see if we can classify them using our model: in other words, can our model at least account for interactive languages and frameworks?

Chapter 5 presents our practical contribution: a tool to support causal understanding in interaction programming - *Causette*. Encouraged by the interview results, literature review from Chapter 1, and some deepenings on the issue in Chapter 4, we think interaction programmers need to understand causal relationships between processes when coding behaviors. We will present a literature review that inspired the tool's design in the field of code visualization for editors. As we will see, the problem of understanding causal relationships in an interactive program is made all the more problematic because current Integrated Development Environments (IDEs) work by splitting the code's causal chains over several files. It becomes difficult to trace the causal chain that explains the behavior of a component from file to file. The tool uses four design principles and four interaction techniques for a graphical and textual editor. An evaluation of the tool is presented, with encouraging results.

Contributions

We sump up here our contributions and will flesh them out in conclusion:

1. An investigation of specific challenges that arise in interaction programming (study with professional programmers) to flesh out more details about the importance of "causality".

2. A novel state of the art, reviewing explicit models of interactive computing in theoretical computer science.

3. Through an epistemological lens, arguing that we lack and need a mechanistic explanation of interactive computing.

4. A proposal for a minimal interactive execution model. It also suggests a concept of interactive completeness/expressiveness.

5. A practical, interactive tool to support interaction programmers in the causal understanding of interaction code through four interaction techniques in a code editor.

Publications and talks

We present the list of publications and talks made during these three years of Ph.D. experience. For the articles whose contents are partly presented in this thesis, we indicate the corresponding chapter or section.

- **Chapter 4** Vers la complétude interactive: exigences pour une machine abstraite orientée interaction: Toward Interactive Completeness: Requirements for an Interactive Abstract Machine. In *32e Conférence Francophone sur l'Interaction Homme- Machine (IHM '21 Adjunct), April 13–16, 2021, Virtual Event, France.* ACM, New York, NY, USA, 6 pages.

  https://doi.org/10.1145/3451148.3458644

- **Chapter 5** Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor. In *30th International Conference on Program Comprehension (ICPC '22),May 16–17, 2022, Virtual Event, USA. ACM, New York, NY, USA, 12 pages.* https://doi.org/10.1145/3524610.3527885

- **Chapter 3** Computers as interactive machines: Can we build an explanatory abstraction? **(forthcoming)**. *Minds and Machines.*

- **Chapter 2** Modelling Interactive Computing Systems : Do We Have a Good Theory of What Computers Are? **(forthcoming)**. *Problems in Philosophy of Science.*

- "Modelling Interactive Computing Systems : Do We Have a Good Theory of What Computers Are?". *6th Conference on Philosophy of Informatics, Frontiers of philosophy of computing and information, December 16-17th, 2022, Varsaw, Poland.*

- "Why Semantics Are Not Only About Expressiveness. The Reactive Programming Case". *5th Conference on History and Philosophy of Programming (HaPoP22'), 13th June, 2022, Lille, France.*

Chapter 1

What is interaction programming?

To begin with, we need to get specific insights on interaction programming practice, especially regarding how programmers deal with causal orchestration in code. These insights will feed our theoretical and practical contributions. The target of interaction programming is the programming of *interactive behaviors*, which is a broader notion than the programming of effective procedures [157, 202, 207, 286].

In a narrow and traditional sense, programming means that one provides a machine with a description for an *effective procedure* or algorithm [135] This description is written in some language that follows a specific grammar. The machine interprets the language in question in a reliable and deterministic way. The Turing Machine provides a model to think of such a specification [97, 275].

In a broad sense, programming specifies, by means of a language, the *behavior* of a machine that can interpret that same language [182]. For example, one wants to program a widget notifying an email that should be displayed for 12 seconds, no more. One wants that whenever the ambient light decreases, the light of the laptop screen decreases as well. The timed email notification and the adaptive brightness are behaviors in the following sense: they describe a reaction that can be triggered at any time by an external event.

We will present in this chapter the results of interviews with 12 professionals specialized in interaction programming. The aim is to understand interactive systems from the programmers' point of view and to collect data on possible specifics of the interaction programming activity, especially the kinds of challenges that programmers face and how they reason about them. Then, we present a literature survey on the programming of interactive behavior. This involves (i) surveying

work targeting the conceptualization and modeling of interaction programming, (ii) looking at the landscape of languages and frameworks, and (iii) summarizing the programming concerns addressed in the literature.

1.1 Interviews with professional programmers

In order to address our epistemological problem, we chose to get insights from practice, investigating how interaction programmers describe their activity. We aimed to gather specific information on the programming and debugging of interaction code. We adopted a qualitative approach [296, 298], and followed a Case Study Research methodology [298].

1.1.1 Participants

The study involved 12 professional programmers of interactive systems. We would have liked to embrace also practices related to interaction programming involving the manipulation of synchronous systems. Unfortunately, all our participants were used to asynchronous systems, with a majority being specialized in the programming of interfaces. Although we will comment on that aspect in section 5.8, we think this restricted sample of practices does not prevent us from covering numerous and significant issues of interaction programming. The application domains of the programmers were the following: Web, Air Traffic Control, Drones, Music software, HCI design, Mobile phones, and Video.

4 of the participants hold a Ph.D. degree in HCI, 4 had an HCI engineering Master's degree, and the other four had a Computer Science Master's degree. Experiences as a professional developer in interactive programming ranged from 2 to 20 years: 5 participants had up to 5 years of experience, and the other 7 had at least 10 years of experience. 10 out of 12 were using the Agile methodology during the development. We intentionally chose professionals as opposed to students. The choice between professionals and students as subjects is debated in software engineering. The topic was delineated in a recent paper from 2021 presented at the International Conference on Code Comprehension (ICPC) [91]. Feitelson presents the issues with students as follows:

> "They (students) may not have fully ingested what they had learned, or hold misconceptions regarding what they have learned. They may not know of commonly used tools or use them ineffectively. They lack practical experience, which makes it harder for them to find and focus on the heart of the issue. Their academic orientation may be misaligned

> with the needs in industry. On the other hand students may be more consistent in following instructions, rather than trying to cut to the core in whatever way (including violating the experimental protocol). In addition, the dichotomy pitting "students" against "professionals" is overly simplistic. Students may have had professional experience in their past or work in parallel with their studies. Graduating students are very close to novice professionals.

We chose to work with developers with minimal experience to get insights on diverse projects in diverse languages and frameworks. 4 of the participants had been working on 2.5 years-long projects in teams involving up to five developers. 2 others were involved in long-term industrial projects (more than 5 years), involving numerous teams with 40 persons on average. Finally, one participant had been partaking in a long-term academic research program, and one participant working for start-ups was used to 3-months long projects. More on project scales can be found in Table 1.1, where interviewees' profiles are detailed.

The participants were using the following languages or frameworks (number of participants in parentheses): Java (6), Python (6), C++ (6), HTLM and CSS (5), Djnn/Smala (3), Qt (3), Animate (2), JavaScript (3), Objective C (2), Java Swing (1), JavaFx (1), QML (1), WPF(1), C# (2), XAML (1), Flash (1), Flex (1), Djnn/Java (1), Perl (1), Rust (1), Unity (1), SQL (1), QML (1), PHP (1), Caml (1).

	Age	Gender	Training	Field	Domain	Experience (years)	Project scale	Languages/frameworks	Methodology
I1	30-40	M	M.S. in Computer Science	Academia	Air traffic control	20	2,5-year-long digitalization of local ATC, less than 5 developers	Djnn Java, Djnn Smala, Html, CSS, Python, Qt	Agile
I2	25-30	F	Ph.D. in HCI	Academia, international company	Air traffic control	5	Long-term (over 10 years) dev. of the Integrated ATC Suite, 300 persons	C++, Java, Java Swing, JavaFx, Html, CSS	Agile
I3	25-30	F	M.S. in HCI	International company	Web	1	3 years dev. of an international airline's website, 50 persons	Html, CSS, Javascript, Java	Agile
I4	25-30	M	M.S. in HCI	Start-ups	Web	3	3-month-long web project for start-ups, 3 developers	Java , Objective C	Agile
I5	30-40	M	Ph.D. in HCI	Academia	Music software, drones, aeronautics	10	15-year-long project for music composers, 3 developers	CSS, Html, Djnn Smala, JavaScript, Java, Python, Qt, Animate	Agile
I6	30-40	M	M.S. in HCI	Academia	Air traffic control	20	Long-term (over 10 years) public applied project for ATC	Python, Java, Djnn Smala, Flex, Flash, Animate	Agile
I7	30-40	M	M.S. in Computer Science	Private company	HCI design	15	Up to 2-year-long project, up to 5 developers	WPF, Csharp, XAML, Objective C, Qt, QML	Agile
I8	over 50	M	M.S. in Computer Science	Academia, international company	Video, mobile phone	20	Long-term projects, 50 developers	C++, Perl, Python, Rust	Agile
I9	25-30	M	M.S. in HCI	Private company	Medical imaging, desktop app, cloud	4	Up to 5-year-long projects, 2 developers	QML, Qt, C++	Agile
I10	25-30	F	M.S. in HCI	Private company	Aeronautics, drone	3	Up to 5-year-long projects, 30 developers	Unity, C#, C++	Agile
I11	30-40	M	Ph.D. in HCI	Academia	Automobile, haptic technology, digital health	10	Up to 5-year-long projects, 2 developers	Caml, C++, Java, Python	-
I12	30-40	M	Ph.D. in HCI Science	Academia, private company	Web, museums	20	Up to 1-year-long projects, 6 developers	PHP, Javascript, HTML, CSS, SQL, C++, Python	-

Table 1.1: Demographic data and interviewees' experience

1.1.2 Method

We conducted the study following the principles of contextual interviews [119]. A contextual interview occurs in the context in which work is being done. It is usually carried out as the work is being done. However, due to the Covid situation that affected the thesis, we had to adapt and relax the requirements for the contextual interviews: we had to find a way to mimic the standard conditions. Indeed, the interviews could not occur at the interviewees' usual workplace. 6 of the interviews had to be online. For those 6 home office workers, even though they could share their screens, it was not the same as observing them for real in an authentic working place. The remaining 6 interviews were in-person, but due to Covid restrictions, the interviews had to take place in a large and ventilated room, not in the interviewees' office. When in person, the interviewees were asked to bring their work laptops. We had then to help and rebuild a working environment artificially. We thus use the critical technique incident [59]. Thereby, the participant had to set up some working environment and then remember current issues (solved or unsolved) encountered in the code. The interviewee and interviewer could then progressively be immersed in a virtual work environment and could look at code together. We describe in the following subsections the participants, the method of data collection, and the data analysis.

Each interview lasted between 45 minutes and an hour. The purpose of the interviews was to identify any specific issue or challenge associated with the understanding and debugging of interaction code. As stated in the introduction , a way to conceptualize the core task of interaction programming is to see that activity not as much as the design of a computation-oriented system. It might be worth considering it as the description of a causally-orchestrated system. We wanted to gain more insights on that aspect. However, we did not ask the participants questions about causality per se, as the participants may not have been familiar with this notion and remained neutral, only asking questions about encountered challenges and bugs related to causality.

After asking the participant demographic questions, we began the interview. We systematically began by asking participants to describe their work environment and development process. This included questions about collaboration with designers, the kind of mock-ups used, and habits in terms of IDEs and debuggers. We then asked each participant questions on the problems they had been confronted with when coding by asking them to remember a particular, recent difficulty, inspired by the critical incident technique [59]. When the participants evoked a problem, they were invited to provide us with a corresponding concrete case and a code snippet. Each time participants presented an example of what we would have labeled "causality issue", we invited the participants to explain the case in depth. The last part of the interview focused on the needs of the programmers, the kind of

tools they needed or solutions they were imagining, and engaging in prototyping. We asked the participant to imagine what kind of representation, information, or tool would be helpful and how one would interact with it. We leave the results of prototyping for later, as this part is too specific at this introductory stage and has served as a basis for design principles in Chapter 5, where we present our tool.

All the interviews were recorded, adding up to more than 14 hours. They were manually transcribed and are attached in Annex B. When quoted in the following, the excerpts are translated from French to English but the complete transcripts are left in the original language. Two collaborators and I analyzed all transcribed interviews by interpreting and tagging them to classify the type of problems encountered by the participants. We then compared the three sets of tags to identify common analyses and came up with 14 labeled problems 1.1.3. From that, we propose a more detailed analysis with a general interpretation and five key issues to which the most recurring tags refer 1.1.4.

1.1.3 Results

We wanted to identify within the transcriptions the "problems" mentioned by interaction programmers. We were looking for statements by interviewers about things hard to code, understand, or debug — generally, any statement about a challenging aspect of their activity. We summarize the key takeaways from our interviews in the following.

The raw results are the recordings and transcriptions. The results we present here have gone through a first interpretation filter, namely, a process of tagging. Three interpreters went through the transcriptions to identify problems described by the interviewees: first, underlying excerpts of interest and then tagging them with an interpretation. The interpretation proposes a more generic issue characterizing the excerpt of interest.

Interpreters A and B defined 25 and 21 tags, with 14 in common. Interpreter C defined 5 tags, all common with the 14 previously mentioned. To count a tag as shared by both interpreters, we needed some consensus estimate to evaluate inter-rater reliability. To ensure the interpreters referred to the same phenomenon or problem using a given tag, we checked whether the tag had been attributed to the same text excerpts (plus/minus one sentence length). If a match was observed, we counted the tag as shared.

A detailed overview of the 14 tags, the variety of cases subsumed under them (with reference to the interviewees), and the number of occurrences per case are presented in Table 1.2 The general definitions for each tag proposed were agreed upon after discussion. A sum-up of their overall occurrences is summed-up in

Common tags	Mentioned cases	Occurrences
Order	Investigating in what order messages/events are received (I7, I10)	3
	Understanding a sequence of events (I7, I10)	2
	Understanding in what order files are instantiated (I12)	1
	Difficulty to understand execution order (I1)	1
Event	Understanding what triggers an event or propagates an activation (I1, I3, I5, I9)	4
	Knowing when an event arrived (I7)	1
	Different event recognition across frameworks, platforms and devices — a swipe or double click (I2)	1
	Transforming a signal into an event for an application (I5)	1
	Understanding the physicality of an app event (what signal, frequency) (I5)	1
	Knowing the activation state of an event or signal — present or absent (I7, I8)	3
Debug	Heavy methods like prints rather than use of fine-grained tools (I1, I4, I12)	3
	Non optimal trials and errors to adjust graphical or haptic properties (I1, I2, I6, I8, I9, I10, I11)	8
	Identifying the level of the problem — low or high-level (I8, I11, I12)	5
	Bug reproducibility (I8)	1
	Getting an answer to a "why question" (I7, I8, I9, I10, I11)	13
	Erratic search for hardware issues (I8, I9, I11, I12)	4
	Finding out dependencies between processes or files in IDE (I12)	1
Animation	Juggling between textual code and graphic tool (I6)	1
	Hard to design physically realistic animations (I2, I3)	2
	Adjusting speed (I1, I2, I8)	3
Finite State Machines	Checking what event triggers a transition (I1, I2, I5, I4, I6, I7)	6
	Checking in what state the system is (I1, I2, I4, I5, I6, I11)	6
	Figuring out what the allowed transitions are (I2, I4)	2
	Struggles with embedded FSM (I4, I6)	2
	Forgetting a transition (I1)	1
	Masks complexify the debug of transitions (I1, I11)	2
	Tedious specificiation of obvious transitions (I1)	1
Architecture	Articulating different physical devices together (I5, I6, I8, I9, I11, I12)	10
	Dealing with the MVC model (I12)	1
	Hard to switch a mode of user action into another (I3)	1
	Writing an animation and describing an interaction on a same object (I1)	1
	Portability issues (I3, I4)	1
Naming	Getting the right identifier and complete path name of a graphic or sound object (I1, I6, I9, I11)	6
Synchronization	Synchronizing frame rate with data reception (I8, I11)	6
	Minimizing latency (I8)	2
	Communication between threads (I12)	1
	Sleep issues (I5)	1
	Synchronizing multiple devices (I9, I11)	2
	Synchronizing sound and animation (I5)	2
	Synchronizing object creations (I1, I10)	1
Code import/Integration	Juggling with several frameworks (I5, I6, I9, I10, I11, I12)	6
Transduction	Adjusting reception of signals and frequency (I5, I8, I9)	6
	Adjusting modulation curve (I5)	2
	Hardware reliability — graphic card, sensors (I8, I12)	2
Dynamics	Ensuring dynamic creation of object (I1, I10)	2
	Initialization issue (I1, I10, I12)	6
Physical time	Adjusting delays and timings (I1, I2, I5, I6, I7, I8)	8
	Adjusting display time and perception time (I1, I6, I8)	3
	Getting access to clocks (I11)	1
	Reference to timer to ensure initialization (I1)	1
	Adjusting frame rates (I8)	1
	Juggling with conversion from timestamps and frame rates, to absolute time (I6, I8)	2
	Knowing exact time of data download (I11)	1
Tests	Tedious scaling-up with real size data (I3, I4, I7)	4
	Ensuring the app will scale on every device (I3)	1
	Heavy tests (I8, I11)	2
	Ensuring the app works with different internet services (I4)	1
Development process	Coordinating workflow and practices between designers and coders (I1, I4, I10)	4
	Tedious tests to check that animations behave as expected (I1)	1
	Lack of documentation to design realistic animation (I2, I6)	2
	Lack of documentation to master physical devices (I11)	1

Table 1.2: Detailed overview of common identified tags and subsumed cases

General problem	General definition	Total occurrences
Debug	Limits of traditional debugging tools	35
Finite State Machines	Issues related to the understanding and writing of a finite state machine	20
Event and order	Issues related to sequences and triggers of events	18
Physical time	Any issue related to the use of timers, clocks	17
Synchronization	Problems with synchronization between events	15
Architecture	Issue about program, file or code structure	14
Transduction	Issues related to the settings of physical devices	10
Event	Problems to understand the triggering of an event	11
Dynamics	Problems with the creation/initialization of objects	8
Tests	Limits with the test procedures to guarantee the app's correct behavior	8
Development process	Issues related to methodology or the articulation between design and code	8
Order	Troubles to understand execution order and/or sequences of events	7
Animation	Difficulties to adjust animations on GUIs, to make them behave as expected	6
Code import/Integration	Difficulties imposed by the mix of several languages and frameworks	6
Naming	Bugs related to path and names for a variable	6

Table 1.3: Sum-up of the general issues, their overall definition and total occurrences

Table 1.3. Because the "order" and "event" issues are very close, we ranked "event" and "order" within one category. As a result, certain tags stand out: "debug" (35), "finite state machines" (20), "event and order" (11+7 = 18), "physical time" (17), "synchronization" (15), "architecture" (14), "transduction" (10).

We are fully aware of the benefits and limits of such methods. On the one hand, we could get qualitative feedback on programmers' experience and gain insights. On the other hand, the pitfalls are related to the reliability of the tags. We cannot assess whether these tags were reliable among the different researchers. There are ways to safeguard and measure rigor in qualitative evaluation (see [270]). One way is inter-rater reliability calculation (e.g., using Cohen's kappa). Without measures like these, we cannot correctly evaluate the results' validity. It would be left to future work to guarantee with quantitative methods the validity of our tags. However, such attempts to quantify the reliability of our qualitative feedback would make sense only with many more interviews, which was beyond the scope of our work.

1.1.4 Analysis

The 14 listed types of problem share some common features, and we propose to break them down into five more general concerns. By doing this, as previously stated, we do not assess any quantitative measurement regarding the reliability of the tags, and we do not state that our breaking down of the tags is the only

available interpretation. We only argue it eases the reading of the results and supports insights into guidelines for interaction programming tools. We leave open to discuss alternative interpretations of our results. The major lesson to be drawn is that digging into interaction programmers' practices enhances challenges concerned with the interconnection of physical and software processes and numerous behaviors, often split across devices, programs, files, and frameworks that require demanding orchestration. Of course, programmers are confronted with algorithmic problems (e.g., sorting algorithms mentioned by I7 and I8), but we want to focus on and detail the existence of another set of problems that is of similar importance in interactive programming practices. Key issues are presented in the following. They are relevant to guide the design of a supporting tool, as we will carry out in Chapter 5.

1.1.4.1 General lesson: more insights on non-algorithmic problems

The search for causal explanations is ubiquitous, as revealed by the cases covered by the *order* and *event tags* where identifying an event source or a sequence of events is at stake.

It is also revealed by the numerous issues related to the causal structure of an FSM (e.g., which transition leads to which state? what are the possible transitions from a state?).

A task that is often described as "unassisted" or "unguided" is the inspection or adjustment of the very low level of the system, involving hardware and various sensors/transduction devices. When an error occurs, interaction programmers often ask themselves *at which level* the problem is located, at a low or high level (I8, I11, I12). At the lowest level, the problem can depend, e.g., on a defective signal or an unreliable hardware element.

Further evidence that interaction programmers can be in the grip of the system's physicality is that they also struggle to get *reproducible bugs* (I8). Because bugs are difficult to reproduce, they are hard to understand and solve. If one tests the system again at another point in time or on different hardware or with different sensors or introduces a breakpoint, the whole system can behave differently.

Physicality is also a preoccupation expressed in concerns about *physical time*. Participants use the notion of *physical* time to make it distinct from *logical* time. Physical time refers to quantifiable absolute time, which is required to adjust the timing or a delay using a timer or a clock. The tags *transduction* and *synchronization* point towards the same concerns with the physical dimension of the programmed interactive system. I8 could not correct his program without finding out that the graphic card was out of use, and I11 had often been faced with faulty sensors.

Those challenges are familiar to interaction programmers, and the proposed tag list may look almost trivial to practitioners. However, as we will observe in Chapters 2 and 3, those aspects have not yet been integrated within a general model doing for interaction what the TM did for computation. Let us look closer at this analysis by going through 4 takeaways that allow us to cover the 7 most recurrent issues (in red in Table 1.3). We propose transversal themes: *traditional tools limitations*, *state and transition mess*, *tricky chain of events and variety of error causes*, and *tedious source code search*.

1.1.4.2 Traditional tools limitations

The more recurrent general issue was debugging, more precisely about the limits of traditional tools dedicated to non-interaction problem-solving. To understand their problems, participants were all using classical debugging tools such as log printing and breakpoints: *"A breakpoint that allows me to go up in the stack and see the calls. I'm sure it's been there, but why? Or conversely, I put a breakpoint to take a step-by-step tour."* (I7). However, they pointed out the limits of such methods regarding interaction code. Interviewees 7 and 8 explained that introducing logs could change the course of execution and that they were often confronted with the non-reproducibility of bugs: *"The problem is more complicated with a bug that we can't reproduce. Every time we try to test this case, it won't happen again [...]. We don't understand what event, what is the sequence of events that leads to this erroneous case. You can use logs, put as many as you can [...], risking bugs with the debug, hoping that it doesn't change the events"* (I8). All participants reported using trial and error heuristics but did not consider them proper debugging methods. For example, I9 called them "tricks": *"when I need to see if there are overlapping components, this is what I do: I enclose my component in a very visible rectangle in a color that pops up. It's obviously a bit of a trick"*.

1.1.4.3 States and transitions mess

The second major issue was the causal understanding of Finite State Machines (FSMs). 8 participants developed at-length issues concerning state machines in interaction code and said they were used to reasoning on paper or code with FSMs. A participant working in web application development stressed that FSMs are an ideal language structure to guarantee the behavior of an application. However, he added that they are time-consuming as soon as they involve multiple states (I4): *"In reality, you don't have the time, so you code in a hurry, you add fields in tables, you end up with booleans, you add statuses. The notion of condition and how you implement it in a certain way is difficult. And also, as the app grows, you need*

to add new transitions incrementally, and it's a mess. FSMs are safer to avoid confusion, but they are hard to use on very large applications, where there are 50 transitions." (I4). The fact that FSMs are convenient but hard to read when they get complex was brought up by another participant (I5). He mentioned complex FSMs that require several actions on the same object ("click", "long click", and "double click"; or a drag&drop that can replace one value with another and be canceled by a click). SwingStates [10] was mentioned by two participants (I2, I4) as a useful notation to understand FSMs. In terms of debugging, the most common method was code print statements. The information participants were looking for is causal: *"to check that we are in the correct state"* (I3), *"here we are not in the state we want"* (I6), *"why, what state am I in? what brought me here? And if I am in such a state, why? Maybe I didn't catch the event when I thought it was going to happen?"* (I5).

1.1.4.4 Tricky chains of events and variety of error causes

A significant amount of problems evoked were related to the difficulty of understanding the possible sources of the modification of a variable and what values it takes. What makes this even harder to understand is the interconnections of numerous devices and software processes that determine the behavior of a graphical component. Two participants used the phrase "chain of events" to refer to their need to figure out what has happened in their application. An extra level of difficulty in identifying an error arises when several programmers collaborate and need to make different parts of the code communicate, calling upon several peripheral devices and sensors (I11).

A common concern expressed by interviewees was the need to figure out the answer to a broad "why?"-question to understand why the programs did not behave as expected. The interviewees insisted on the diversity of the causes of interactive bugs, ranging from implementation issues and the settings of physical devices to issues concerning the behavior of a graphical object. For example, when programming GUIs, a graphical object might behave unexpectedly (*"objects that move, without I knowing the event that triggered them"* (I1)), or animation or graphical layout is not correct (*"why are my components not displayed as expected?"* (I7). Interviewees explained that tracing the source of such an unexpected behavior is tedious. The reason is that interactive behaviors involve multiple layers from which the error could stem: *"Is it a quality problem? Is it a quantitative problem? Is it an event problem? A timing problem? Or is it something low-level? "* (I8). For example, the unexpected behavior of a GUI component could be due to incorrect signal processing at the physical layer (I5, I8), wrong information parsing from a software bus (I5), or incorrect renaming of a text variable (I1).

1.1.4.5 Tedious source code search

The challenge of mastering the orchestration of behaviors across several physical devices is made even harder because of behavior descriptions split across files. In other words, it is hard to grasp the overall code describing a behavior without having to spend time navigating between files. Seven participants described the search in interaction code to be particularly intricate, to the point where they found some limitations in search tools (e.g. "jump to definition"). They fleshed out two reasons for this. First, they pointed out the complex file hierarchy involved: *"this instantiation of files hierarchy...it makes me cry. You trace, you make prints to see 'aha this is where it's loaded', or you make mistakes in your code, creating bugs, so that the console tells you in which file it's wrong"* (I11). I12 more specifically mentioned the Model-View-Controller structure on which interaction code is often based: *"I don't like the way the code is split up in many files. Likely, someone who's done MVC programming for a very very long time can find his way around. But I have a lot of trouble and it's really by habit, by getting into a project that you're going to acquire a knowledge of that project but that's ephemeral. The risk is that if I get back into the project later, I won't remember. There's no trace."* Second, even if they could use search tools to navigate the code and find the definition and various instantiations of a variable, it was not sufficient to give them the mental picture of the causal chain, from a definition to an instantiation.

1.1.5 Interview sum-up

The results of the interviews give us first insights into interaction programming practices. As we pointed out, the recurring issues encountered point in the same direction: interaction programming has important aspects that cannot be broken down to algorithmic problem-solving. However, this does not suggest that interaction programmers are not faced with algorithmic problem solving [1]: the point is simply to consider the other sets of non-algorithmic problems at stake to gather a comprehensive view.

A part of their activity appears in the description, adjustment, and understanding of references to physical events and processes. A symptom is the number of problems with classical debugging tools dedicated not to interaction programming but to computation (e.g., correcting loops, inspecting results of calculations, for example). The tag *debug* ("Limits of traditional tools") covers most mentioned issues indeed. The result is reminiscent of an argument made in Salvaneschi's work [245, 250], calling for a new debugging paradigm adapted to interaction program-

[1] I7 and I8 commented on struggles with sorting algorithms.

ming [2]. A recurrent struggle found within the debug category was that interviewees spend a lot of time asking "why" questions (13 occurrences in total, among I7, I8, I8, I10, and I11) to find the origin of abnormal behaviors of the programmed system. This is reminiscent of the conclusions drawn by Ko and Myers motivating the design of the WhyLine tool [136, 205] [3]. Many participants introduce their own term to refer to bugs that they believe do not have a computational solution but bugs that can only be solved using human perception: this is what some call "haptic" or "visual" bugs (I1, I2, I6, I8, I9, I10, I11), or what they call "adjusting display time to perception time" (I1, I2, I8). For example, I2 commented on issues with animations, telling she spent a significant amount of time adjusting the behavior of the animation to make it pleasing to the eye.

1.2 A few dedicated models

We will now review how these issues have been conceptualized and addressed in the literature. More precisely, we review:

- Available **models and concepts** of interactive systems, as formulated by practitioners
- The overall landscape of **languages and frameworks** supporting interaction programming
- Well-known addressed and remaining **challenges in code** stated in the literature.

As already recalled, our perspective on interactive systems is not that of the search for a socio-technical theory of human-computer interaction [4]. We focus on the specifics of interactive systems *from a software and hardware point of view*. The aim is to build a model that would do for interaction what the TM does for computation: a model providing the basic building blocks that allow describing a language executed on it and intuition about its execution. This, in return, should provide an account of what makes an interactive computing system *programmable*.

[2] *Interactive programming* in our extended sense — Salvaneschi talks about "reactive programming" in his own words

[3] The WhyLine is a debugging tool developed by Ko and Myers [136, 205], targeting answers to why-questions asked by the programmer.

[4] Among interaction theories, some focus on explaining human activity and behavior when interacting with technology, as can be found, for example, in the Generative Theory of Interaction proposed by Beaudouin-Lafon, Bødker and Mackay [23] or in Hornbaek and Oulasvirta's work [120, 219].

As stated in the introduction, we are looking for an abstract and general model accounting for interactive software and hardware as a programmable system. It targets the general properties required for interaction programming at a level of abstraction akin to the TM when describing a computation. To build it up, we, therefore, need to identify the essential bricks of a general programmable interactive computing system. Therefore, when investigating the literature with that goal in mind, we find material among software engineering models rather than general human-computer interaction theories. We will focus on four relevant proposals. They are relevant because they identify the minimal abstract components to program an interactive system.

From the comparison of these four proposals, common building blocks for the design of an interactive computing system can be identified. These design blocks have their counterpart in the kind of language allowing for programming the system. In other words, they have their counterparts in the expressiveness of interaction-oriented languages. Their synthesis and interview results are combined within an execution model dedicated to interaction, as proposed in Chapter 4.

1.2.1 The architecture of reactive *computational artefacts* — Suchman

Suchman's work on the architecture of interactive systems is close to our work [271]. We also refer to a recent survey and analysis of her work [256]. At a time when reflections on the interactive paradigm were emerging (more on the so-called interactive computing paradigm in Chapter 2), Suchman introduced the concept of *computational artefacts* in order to address the nature of "interaction". Computational artifacts refer to the kind of computing devices designed to react and be part of activities in which they are used. To Suchman, not all computing devices can be said to be "reactive" (or *interactive* given the terminology used in our thesis). She points out that the reactivity of computational artifacts relies in "the availability of interrupt facilities whereby the user can override and modify the operations in progress" [271]. By contrast, computer devices used in batch processing modes are not reactive. Minimal features of these artifacts' architecture are fleshed out, and we propose to sum them up in the following. The features suggest the founding blocks used later in our proposal (Chapter 4).

Suchman insists on a first building block: **time**. For the reactions of the device to be experienced as immediate and the device to be experienced as 'interacting', or 'real-time', a response time adapted to state-changes in the environment is needed. Suchman points then at the necessity of "manual intervention" [243] or **interrupts**. Interrupts support the very possibility of interactivity from a physical point of view: "the availability of interrupt facilities whereby the user can override and modify the

operations in progress" [271]. Suchman talks about interrupts, which are a common way to implement the communication between computing processes inside the machine and external processes in the environment[5]. To Suchman, interactive computing devices do not simply execute a pre-given set of 'commands' in a predefined sequence. Rather, a program awaits the occurrence of certain external events to execute a 'command' (in source code expressed as, for example, 'when event A arrives, do X'). "Recombination of programs" or **dynamic change** are a third essential block. The device can react but also change behavior in response to changes in the external environment, especially to users' actions. It may also modify its own 'code' and do so in 'real time'. Reactivity does not rely on increased operational speed but on the capacity to change behavior quasi-dynamically as environmental conditions change. Furthermore, 'programs' can 'call' other program, even a program residing on a remote device or hand over 'control' to another program.

1.2.2 Fundamental structure and concept of interaction code — Letondal and al.

Letondal et al. [153] following up on previous reflections [57] analyze discrepancies between interactive and non-interactive software and localize them in the code structure and the development processes. By carrying on that project, they suggested building concepts for interaction-oriented programming, some overlapping with those identified in Suchman's work.

In the authors' words, **"contravariance in reuse and control"** is a core dimension of interactive systems and overlaps somewhat with Suchman's interrupts requirement. There are two ways to deal with control flow in an interactive system: either function calls requiring events or dataflow; or, on the contrary, the flow comes from the outside of the main program. In the latter case, the application receives control from, e.g., input drivers or interactors. Although Letondal and al. do not detail the implementation of control inversion, its hardware counterpart can be precisely what Suchman refers to as interrupts. Then, in interactive systems, the authors enhance that the complexity is in behaviors, that is, in the **change of state** of objects, and not only in computations. **Concurrency** is essential since interactive systems involve concurrent processes such as animation. Finally, programmers, graphic designers and even end-users are involved in producing interactive applications. Therefore, the same object, like a button, designed by a graphic designer and then coded by a programmer, might be manipulated by an

[5]Other mechanisms could be used instead of interrupts, like polling (more details on this in Chapter 4). But the idea remains: there is a need for *some* mechanisms to explain how a computing system can react to the arrival of external inputs during execution.

end-user wanting to modify the font of the button. In other words, the interaction involves what the authors call **"different reuse patterns'**.

1.2.3 The Anatomy of Interaction — Basman et al.

The work of Basman et al. [19] also targets the level of abstraction we have in mind when looking for a model of interaction programming. They propose a taxonomy "for describing the conditions and implementation of interactions". They intend to overcome a gap in the available explanation for interaction that they think is due to a separation between the programmers producing the software and users interacting with the software. How the interaction occurs and what mechanisms support it is left unsaid.

Basman and al. propose to fill the gap and present the missing mechanisms. They identify two phases in the interaction between software and an external event (from the environment). These two phases help flesh out the "blueprint of a system" and some relevant mechanisms. The result is intended to be a sketch of an interactive system with explanatory power. The first phase is called "co-occurrence": "(it) determines what elements of the design are in a configuration in which an interaction that involves them may potentially be initiated". The second phase is called "entanglement": "the temporary coupling and interplay between elements participating in the interaction", "(it) is both a process and an object, i.e., a new element that represents the interaction for the duration of its lifetime".

The authors describe the flow of interaction, starting from what they call the state of the world (of interest), that corresponds to what is called the Model in the Model-View-Controller (MVC) representation of some interactive programs [6]. At the operating system level, the states of the world refer to the device drivers. Then, "co-occurrence" is needed before the interaction starts: "Certain elements of interest must have been assembled in a 'suitable proximity' — this proximity may be physical, informational, or take some other form that makes the elements conveniently available to each other or the user". The authors give some examples of co-occurrences: e.g., a finger touching a screen; a color-picking instrument targeting a particular pixel; a user's gaze tracked by a camera. Any co-occurrence takes the form of a signal containing references to the co-occurring elements emitted as long as the co-occurrence is ongoing. A co-occurrence engine accesses the signal in some "state of the world" document. Then, co-occurrence is externalized. After a suitable co-occurrence has been recognized and becomes a co-occurrence signal, it must be acted upon to initiate an interaction. The entanglement instantiator "maps"

[6] The Model-View-Controller triad was initially presented in [238]. It is often used as a basis for thinking of an interactive system and structuring its code in many HCI applications

detected co-occurrences into entanglements. The entanglement lasts as long as the interaction and represents the interaction as an accessible element during the system's runtime. An entanglement can give rise to further co-occurrences and hence entanglements. As we will see in Chapters 3 and 4, the authors' modeling project is close to the requirements we present for an execution model dedicated to interaction. We think the motivation is similar, as summed up in the introduction of "Anatomy of Interaction": "We believe that the major fault of current approaches to programming interactions is that they do not account for how interactions come to be" [19].

1.2.4 A system engineering model for interactive systems — ICOs notation

In a 2009 paper, Navarre et al. present ICOs, a user interface description language (UIDL) [213]. The motivation is to offer a more systematized model to support and guide researchers and practitioners in the User Interface (UI) domain.

Although UIDLs are a specialized subset among interaction-oriented languages, the paper provides useful concepts on expressiveness for interaction.

The ICOs notation is a basis to think of criteria for interactive expressiveness and to support various system engineering models for interactive systems. The tool supporting the ICOs notation (called Petshop) is a Petri nets-based tool for the design, specification, prototyping, and validation of interactive software. The authors emphasize the expressive power of their proposed notation and make it a candidate to support the description of various aspects of user interfaces, (e.g., WIMP and post-WIMP interaction techniques, the behavioral part of interactive applications).

Six different properties of the language are used to characterize the expressiveness of UIDLs. The more a UIDL *explicitly* express these properties, the more expressive it is.

1. **"Data Description"**: it deals with the language's description of objects and values. More precisely, it involves modeling data emission and reception from input and output ports.

2. **"State Representation":** it refers to the description of states. It is reminiscent of one of the four core dimensions defined by Letondal et al.

3. **"Event Representation":** it refers to the description of events.

4. **"Time":** This criterion was also found in Suchman's work. ICOs delineate that aspect more specifically, addressing two different representations of time: quantitative and qualitative.

   "Quantitative time" represents behavioral, temporal evolutions related to a given amount of time (usually expressed in milliseconds).

   "Qualitative time" aims at representing the ordering of actions such as precedence, succession, and simultaneity.

5. **"Concurrent Behavior"**: As pointed out by Letondal et al., a representation of concurrent behavior is necessary. ICOs' authors have more precisely in mind that interactive systems feature multimodal interactions or can be used simultaneously by several users.

6. **"Dynamic Instantiation":** Dynamic instantiation of interactive objects is a characteristic required for the description of interfaces where objects are not available at the creation of the interface, e.g., the creation of new icons according to user actions. Dynamicity was also mentioned by Suchman.

1.3 Dedicated frameworks and languages

Programming has evolved to make easier the design of interactive behaviors. Because interaction programming has emerged among existing programming practices dedicated to non-interactive programs, different programming styles, programming models, and paradigms (functional, object-oriented, and reactive) are available. A collection of new languages and frameworks exist and have been surveyed in the reactive programming literature [16, 250]. To complete our study of current interaction programming practice, let us focus on how interaction is currently programmed.

Some frameworks used in interaction design rely on languages or frameworks not initially dedicated to interaction, meaning they are not entirely shaped for interactive programming. The current supporting interactive frameworks may be based on different programming paradigms, such as object-oriented programming (e.g., Java, providing an interactive extension, JavaFx) or functional programming (such as ML offering an upper layer allowing the programming of real-time interaction: Reactive ML [167, 168]).

To present an overview of the landscape, finding a criterion guiding a typology is not straightforward. There is a distinction between synchronous and asynchronous, compiled and interpreted languages. Compiled reactive languages like Lustre or Esterel often subscribe to the synchronous hypothesis. Synchronous languages are

based on the so-called zero delay model, i.e., the time that elapses between two clock ticks is considered zero time during which nothing happens in the external environment. In this model, time is logical, and the execution is seen as the system's sequence of atomic reactions (or steps) to input events. It is assumed that outputs are computed together in zero time within a step because parallel components synchronize their reaction steps by the semantics of the languages. This is why synchronous languages are said to be deterministic. For the model to be safe, it is then necessary to check the correspondence between logical time and physical time: is the machine fast enough for the approximation to be satisfactory? It is not obvious, however, that the synchronous/asynchronous dimension should be used to distinguish between languages. Indeed, the synchronous hypothesis essentially serves verification and certification needs. In this sense, a compiler for a synchronous language such as Lustre could support dataflow verification for other reactive languages.

However, those distinctions would end up with many orthogonal parameters and overlapping categories, making the typology too complicated here for our purpose. In order to delineate the landscape, we propose to consider existing languages and frameworks as a spectrum: from the less to the most interaction-dedicated.

Interaction dedicatedness	**Language and framework examples**
Degree 0: Adding an interactive layer	Qt, JavaFx, PyQt, Flapjax, QML-Python
Degree 1: Adapting the functional paradigm	React.js, Elm, Ceu, ReactiveML
Degree 2: Inventing a new conceptual model	Lustre, Esterel, HipHop.js, Garnet, SwingStates, FlowStates, ICON, Smala, Act, Pict

Table 1.4: Programming interaction: languages/frameworks/APIs overview

So, for the sake of simplicity, we propose to distinguish the three following sets of languages and frameworks (with examples detailed in Table 1.4):

- **Degree 0**: Interactive languages, frameworks and toolkits supported by a non-interactive host language (e.g., C supporting Reactive C [37], C++ supporting Qt, Java supporting JavaFx, Python supporting PyQt). That involves no dedicated syntax.
- **Degree 1**: Functional reactive languages [54, 73, 88, 251, 283], which support interaction by adapting the functional paradigm to insert it within the reactive paradigm, with the introduction of side-effects.
- **Degree 2**: Dedicated languages and frameworks, with dedicated syntax and semantics. It involves theorizing and introducing new conceptual models. Statecharts [115], Signal [110], Garnet [207], Esterel [27, 28, 92], Lustre [53],

HipHop.js [29, 30], SwingStates [10], FlowStates [11], ICON [84], process-based languages such as Smala (introducing bindings, connectors, processes) [164] or far less recent Pict [229] Actor languages (introducing messages and actors) [4, 5, 139] are such examples.

At this stage, we cannot go any further in refining the landscape of languages and frameworks. Given a more detailed view of interaction expressiveness, as fleshed out in Chapter 4, we will be able to go back to this typology and delineate it more.

1.4 Interaction programming challenges stated in the literature

To complete the horizon, let us examine specific difficulties encountered to support interaction programming, as found in the literature and reminiscent of the problems stated by the interviewees. To structure this literature review and shed more light on the interview results, we looked at the more recent and quoted survey papers [16, 247, 250, 252], theses, and major research projects in the reactive/interactive field. The struggles encountered in interaction programming are not about computations or computational complexity and help to flesh out the idea that interactive computing needs new models and tools. They are symptoms of interaction programming not being reducible to computation-oriented programming. We propose a few thematic items to present the key insights.

1.4.1 Semantic and syntactic concerns

In the interaction programming community, practitioners often state that computation oriented languages do not ease interaction programming [22, 57, 153, 206, 235]. Programmers of interactive systems face specific challenges due to the interrelated behavior of numerous components and uncontrolled, unpredictable flow of external events [153]. But the basic concepts of today's languages are dedicated to calculations (functions, arithmetic expressions, data structures). These languages often relegate interaction programming to a secondary rank, making it difficult. There have already been semantic upheavals to ease interactive programming. Several textual programming languages provide dedicated language constructs to tackle the complexity. Some languages propose reactive constructs that avoid the need to write code for updating outputs when the inputs of a computation change [283]. Others use traditional object-oriented constructs to describe state-based behavior [10, 11]. Conceptual frameworks, syntaxes and tools have been proposed for user interface development [208], natural programming [209], interaction-oriented

programming [164], or web programming [30]. Easing interaction programming often relies on completing existing frameworks. Object-oriented languages provide support e.g., for event references, with properties in JavaFx or signal/slot mechanisms in Qt. The data-flow concept has been introduced in the functional paradigm [88, 177]. Reactive functional programming has borrowed from object-oriented languages its concept of event [251]. Synchronous languages have been adapted to allow dynamic reactions, such as in ReactiveML, mixing the typical behavioral semantics of synchronous languages with transition semantics to control dynamicity [167]. Some proposed mix data-flow languages with control flow [11, 65]. In other words, at a semantic level, available interaction languages are often forced to combine programming styles or even paradigms. Semantics for interaction programming is pervaded by heterogeneous abstractions, such as events, signals, objects, and functions. With some exceptions, as we stated in Section 1.3, not many languages can be said to be dedicated to interaction programming.

Another problem left at the semantic level is that programmers do not have access to the low level, which constrains how fine-grained and tuned programmed behaviors can be. Remember that reactions to events (user actions like input from the keyboard, the mouse, or the touchscreen, arrival of new data, change of value etc.) are expressed by programs. Some reactions are specified mechanically (the sensation of the mouse "click"), and others electrically (the lighting of a led when the computer is turned on). Programmers need some access to that level of specification, in other words, to the physical low-level of the machine, where access to the drivers and peripherals providing the inputs are available [51, 57, 84, 117, 122, 153, 201, 264]. Existing programming languages are combined with a software library that detects the actions of the users and records programs to be executed on actions. Libraries provide services related to the management of inputs and outputs with the interaction devices, which make them essential for programming interactions, but at the same time hide the low level of interest from the programmers. The issue of low-level access has been brought up many times in the interviews. The interviewees mentioned, for example, the need to have direct access to the frequency rate of a streaming input. This access needed by the programmers to refine the interactive behaviors can be uneasy or impossible because APIs usually hide the desired low-level specification. Another common example is how to write a double click. Without access to the low-level of its implementation, its functioning is opaque: which delay during two clicks actually defines it? This access is even more important because the environment around an interactive computing system can vary in input and output devices: mouse, keyboard, trackball, touchscreen, speech input, small or large display, etc. Therefore, programmers need ways of describing what devices they wish to use and how. As some have phrased it in the literature [57, 153], an interaction programmer sometimes has to turn into a device driver programmer.

Semantically, the available *APIs* for interaction programming are not only problematic because they are vertically multi-layered and opaque. Another problem is that interaction programmers do not program abstract computations but program behaviors dependent on some context of use. Specific needs are needed depending on the kind of programmed behavior and object: it is not the same to program the behavior of a video game character, a plane on an air traffic control's user interface, a video stream, or flight commands. Therefore, interaction programming involves numerous specialized libraries, which do not provide a unified and flexible framework. This has already been labeled as "horizontal multiplicity" [235].

If semantics have evolved to support interaction, syntactic issues are still at stake. For example, the expression of relationships between events and reactions [164] can still be more verbose in some languages and frameworks than others. For example, we can compare two structures that express a coupling between an event and a process: a signal/slot in Python

```
object1.signal.connect(object2.slot).
```

and a binding in Smala:

```
process1 -> process2
```

At the syntactic level, another remaining issue is the description of numerous behaviors in code in reacting to events. The problem has been addressed and famously labeled by Myers as "spaghetti of callbacks" [204]. It turns out that the writing of functions is cumbersome in the context of interaction programming. Today, different strategies are available to optimize the execution of interactive behavior. They provide various execution models: e.g., sequence of instructions, automatic updating of properties, and event listeners.

1.4.2 Execution concerns

If we leave semantics aside, we find in the literature comments on execution concerns. We can at least mention three of them.

First, ***latency*** is a challenge [52, 68, 297]. Interrupt latency is the duration that it takes for a computer to act on a signal that tells the host operating system (OS) to stop until it can decide what to do next. Latency is a synonym for delay; the shorter the latency, the better the quality of the user experience. But latency is not only a qualitative matter and measures of latency have been proposed [52, 68]. Latency involves more specifically three issues: (i) a synchronization issue, (ii) the length of the processing chain from input to output across layers, and (iii) the cost of the display. "Reactive" and "interactive" systems [7] are not equal when

[7] The distinction is presented in the Terminological clarification of the **Introduction**.

it comes to minimize latency. In the case of "interactive systems", the external environment controls the execution, and the system does some *passive waiting*, idling until the environment notifies of a change. This allows lower latency but poses synchronization issues. On the contrary, in the case of "reactive systems" programmed with "(synchronous) reactive languages", the system is actively and steadily checking for changes, following internal clock ticks that drive the execution. Synchronization is in principle guaranteed.

Second, related to ***synchronization challenges***, is the *"glitch" issue* [16, 69]. Without a synchronous hypothesis, it has to be guaranteed that the systems react correctly to update the reactive value in a program. Glitches are update inconsistencies that may occur during the propagation of changes. When a computation is run before all its dependent expressions are evaluated, it may result in fresh values being combined with stale values, leading to a glitch. For example, let us consider the following pseudo-code:

```
2 + a = b
b + 1 = c
```

If the value of a changes, the value of b and c is expected to change. And the value of c should not be recomputed before the update of b. If that is not the case, the values become momentary inconsistent: this is what is known as "glitch". Glitches result in incorrect program states and wasteful recomputations and, therefore, should be avoided by the language. Most reactive programming languages eliminate glitches by arranging expressions in a topologically sorted graph [69, 165, 177], thus ensuring that an expression is always evaluated after all the values that it depends on have been evaluated.

Third, the variety of ***control inversion*** modes influence interaction programming styles and can be problematic. When relying on frameworks, control inversion means an application entrusts its code to the framework for execution rather than executing it itself [129]. Control is thus inversed since it is the framework that is responsible for organizing the various functions under its responsibility. Today, several forms of control inversion exist, depending e.g., on the used libraries/frameworks [8] In the case of a library like JavaFX, programmers provide objects whose methods are executed by the framework (widgets) and also provide blocks of code to be executed when events are triggered (callbacks).

Fourth, the search for ***dynamicity*** has to be addressed to allow richness of interactive behaviors [50, 126, 167, 208, 213, 213]. By dynamicity, we mean adding

[8] For a more fine-grained distinction between a library and a framework, a criterion based on control inversion has been proposed [235]: "a framework (...) is a software library that permanently appropriates the execution control of the application, and allows the execution of third-party code by receiving callback functions and methods."

new behaviors associated with newly created objects at runtime. It is a question of being able to change the components or the parents of a given process at runtime. For instance, when using a touchscreen, each finger moving up and down is a virtual input device being added or removed from the system at runtime. Not every interaction language allows it. In Chapter 4, we will propose a typology of interaction frameworks and languages, taking into account their dynamicity as part of their expressive power.

1.4.3 Understanding concerns

The third set of concerns we identified in the literature, motivated by some findings in the interviews, is related to interaction program *understanding* in the course of *debugging* tasks. The reactive programming literature has proposed a new debugging paradigm dedicated to the debugging of behavior. A claimed motivation is that programming behavior involves specific understanding challenges [136, 205, 247, 250]. It has been commented, for example, by Salvaneschi and Margara [170, 246, 248, 250].

1.4.3.1 Understand the state and the flow of the system

When surveying models and major concerns in interaction programming, we noticed the importance given to the transparency of applications and access to their states. The lack of information on the internal state of applications forces designers to guess it from what they observe [153, 235]. The issue is reminiscent of what Norman had called the "Gulf of Evaluation and Execution" [217]. The so-called "gulf" is about the degree to which a computer system's interaction possibilities correspond to the user's intentions and to what the user perceives is possible to do with the system or some application.

Tools for visualizing program execution have been proposed to address some aspects of this issue. Debugging lenses provide access to the system's state by enabling the programmer to see information about the attributes of interface elements using movable floating windows. ZStep offers mechanisms for understanding the program's behavior by stepping through graphical changes in the user interface instead of through lines of code [278]. The WhyLine allows developers to perceive and interpret the system's state in terms of the actions that produced it. For example, developers have answers to which statement has set an attribute and why a window encounters such change [136]. SwingStates provides a visual depiction of its finite state machines' dynamics, allowing them to be related to interface changes [10]. In some cases, the tools that enable the perception of the program disturb its

flow of control and thus its correct behavior [121]. In others, the perceptible result of the program is difficult to relate to the code that produces it [157].

Understanding not only the states but also the flow of an application is a challenge as well. It is a key takeaway of the interviews, enhancing a requirement already made in the literature for interactive software [109, 153, 245, 247].

1.4.3.2 Restoring understanding across split code

The difficulties in understanding states and flow, are made even more complex because of the nature of APIs and Integrated Development Environments (IDEs). In the context of behavior programming, IDEs add challenges to the writing and understanding of code. Programmers have to overcome the challenges related to code splitting over multiple files. The literature highlights how IDEs lack effective support to browse complex relationships between source code elements. Developers are often forced to exploit multiple user interface components at the same time [138], making the IDE "chaotic" [186]. Time-consuming navigation between files has been studied [40, 41, 240, 241].

The interviewees have commented on the challenge. In the case of interface programming, the Model-View-Controller (MVC) software architectural pattern reinforce the need to overcome split descriptions of behaviors. In that specific case, the programmer goes back and forth between three interrelated components to describe the behavior of the interface: the "model" (managing the data and logic), the "view" (managing the representation of information), and the "controller" (accepting inputs and converting them into commands for the model or view). In general, most approaches in interaction programming rely on code split over different source files, each dedicated to describing interactive behavior at a local level. This breakdown hinders the programmer's ability to understand interactive behavior. As a consequence, interaction code is hard to debug and maintain. This kind of challenge has only been addressed outside the interactive programming community, and we will deepen the issue in Chapter 5, Section 5.1.2.

The semantic and execution issues we surveyed, relevant to feed our theoretical work on interaction, will not be the target of our practical work in Chapter 5. We will focus on interaction program *understanding*, which were the last set of mentioned concerns.

1.5 Summary

Chapter 1 provides an introduction to the specifics of interaction programming. We presented the results of interviews with 12 professional programmers reporting the

major problems they encounter with respect to causality. We gained more insights on non-computational challenges and errors to solve, related to the physicality of the system and the orchestration of devices, signals, events, files and frameworks. Control of time, transduction, understanding the origin of an action, sequences of events, or identification of the bug level (low level or high level), in addition to the mastering of algorithms, are interaction programmers' daily concerns. The issues scale up when the architecture of software and interconnected devices get more complex, involving multiple data streams and files to articulate to produce the expected behaviors.

We surveyed the existing characterization of interactive software and hardware within the interaction programming literature. We finally had a look at existing languages and frameworks and a synthesis of addressed and remaining challenges of interaction programming. Our interviews and the literature substantiated the difference between computational and behavioral programming. In particular, the interviewees illustrated several cases where understanding and writing code involves understanding low-level physical phenomena (e.g., readjusting the signal reception frequencies or the timing of animations) and the connection between events (how it is declared and implemented). We gained more insights on causality issues, and it turns out interviewees and the literature use the concept, with few making it explicit [207, 235]. Indeed, we find in the interviews as well as in the literature the idea that the main need is to understand causal relationships between processes described in the code. This concept is used intuitively without reference to a specific theory of causality. We propose to make it a central concept for a model of interaction programming and (i) to integrate it into an execution model, (ii) to use it to support the understanding of code in a tool.

As we aim at building a model for interaction, we need to determine both what is the (i) core concept of this model - if not computation. We must also identify (ii) the needs motivating such a model - if not only understanding algorithms. We have already supported the hypothesis that causality is a good candidate for the core concept. As for the needs, they generally consist in understanding how interactive behavior is orchestrated, from the code to its physical reality at the implementation level (where the programmer needs to understand in particular the physical dimensions of a signal in order, e.g., to adjust its frequency).

The accounts of interaction from the HCI community we covered in Section 1.2 provide us with building blocks but do not provide us with a general model for interaction, at the same level of abstraction as the TM for computation. The question remains then whether other fields have provided such a model. We will look at theoretical computer science, in which TM originating from computability theory serves as a founding model, to see whether models of interaction have been proposed. It is a fact, as will be presented in the coming Chapter 2, that the

Turing Machine, which initially describes sequential procedural behaviors, has also been extended to account for some aspects of interactive behaviors. For example, many algorithms have been proposed to solve identified issues, such as synchronisation [140] and concurrent access [81]. A theory of mobile communicating systems has dealt with concurrency between processes [183]. Temporal logic has provided a notion of timed automata [7]. More recently, formal proposals have defined a *Reactive Turing Machine* [15]. Some considerations on a warranted paradigm shift have also been suggested [279]. However, and this will be the point of the next chapter 2, the kind of models provided in theoretical computer science is too far away from the concerns and reality of practices stated here in Chapter 1. We will also see later in Chapter 3 that the kind of models formalizing interactive systems in computer science does not provide us with the kind of abstraction suitable to our purpose: they cannot serve as an *explanation* of how an interactive computing systems produce interactive behavior.

Chapter 2

Interactive computing in theoretical computer science

Epistemologists have asked what computing systems are [225, 236, 266]. The question stems from both philosophers of computing, like Piccinini in "Computers" [225] or Rapaport in "What Is A Computer? A Survey"[236], and computer scientists like Smith in "The Foundations of Computing" [266]. Two difficulties lie in the question. First, the evolution of computing complicates the question and any attempt to answer should take recent computing practices into account. In particular, while epistemologists have been interested in conceptualizing what a computing system is for a long time, they seem to have paid little attention to the specifics of interactive computing. Thus, there might be a risk of not offering an adequate conceptualization. Second, as noted by Rapaport [236], many answers to the question distort it by answering the question of what a *computation* is instead. Therefore, they are immediately projecting the field of investigation into the computability theory:

> "A fairly obvious, trivial, and almost-circular definition of 'computer' says that a computer is a machine that computes. The natural next question is: What does it mean to compute? But this shifts the burden of answering our question away from what computers are to the topic of what computation is. Many of the objections to various theories about computers are really objections to what counts as a computation."

When looking for explicit theories of interactive computing systems outside the field of HCI that we previously surveyed in Chapter 1, we find them within theoretical computer science, framed in terms of "theories of *interactive computation*".

In this chapter, we examine whether models of computation for interaction allow us to answer the question of what an actual (necessarily interactive) computing system is. We propose a literature survey in theoretical computer science where one can find explicit proposals for a model for interactive computing. We show that the formal modeling of interactive computing systems has been brought down to whether the new interaction models are reducible to Turing Machines (TMs).

Questioning the theoretical bounds of the Turing Machine in computer science when faced with the existence of interactive devices has been explored at least since Milner's work on communicating and mobile systems [182, 183]. At first, an interactive computer system was defined as a system where several threads execute instructions in parallel while being able to synchronize and communicate at certain moments of the execution. Since then, the characteristics of computer systems have continued to evolve, and by "interactive" we refer to a broader set of properties that can be grouped as follows: the ability to continuously react in time to external events that modify their course of execution.

To the best of our knowledge, there are three areas where interaction models are framed. In all of them, the comparison between TMs, oracle machines, and interactive system models is systematically at stake. These areas are namely work on:

- concurrency by Milner and his followers [183, 184],
- Reactive Turing Machines and
- interaction as a new computing paradigm.

For each of the three identified models,

- we present the motivation behind it,
- we sum up its account for interaction (its expressiveness and possible equivalence with a TM)
- we identify how it has been used, and criticized [15],
- and we suggest further issues regarding a theory of interactive computing [103, 286, 290].

We will show how these formal approaches cannot provide an answer to the epistemologist for two reasons. On the one hand, these models of computation have focused their attention on whether interactive models are reducible to models of classical computation - par excellence, the Turing machine. Proving (or not) that

an interactive property can be formalized as a computational property in the classical Turing sense does not answer the question of how an interactive property can exist and be the object of execution. On the other hand, and this is a correlate, these models do not propose a basis for a mechanistic explanation of how interaction is made possible. With only formal models of interactive computation, we might run the risk of not offering an adequate conceptualization of interactive systems.

In this chapter, we only survey the *explicit* proposals for a general theory of interaction. In Chapter 3, we will also cover formal models dedicated to properties required to account for interactive systems (e.g. measurement of physical time, streaming data). But these models are not integrated within a general theory of interaction.

2.1 Milner: interactional vs. computational

2.1.1 Motivation

Milner was the one introducing the concept of interaction in computer science. His famous Turing Award speech [182] provides a sum-up of his motivations. Milner was concerned with the logical foundations of computing inherited from Turing. He was preoccupied with the idea that computing practices had evolved since the birth of computing, notably in terms of architecture. He took the possibility seriously that the logical foundations dating back to the thirties may not match the growing challenges of his time and may require additional concepts.

Milner pointed out [184] that the logical foundations of computing offered by Turing [275] predated the first physical computers and that computer science is grounded in logic and engineering. On the engineering side, computer science had inherited from the pioneering work of von Neumann [100, 131]. Only one thing could happen at once in an early von Neumann's computer. Nevertheless, there was more to computing than von Neumann's architecture [13, 184]: a growing interest in dealing with concurrency in the sixties and seventies made sequential programming less warranted. Therefore, to Milner, the logical foundations of computing were to evolve. The main flaw of these logical foundations was the reduction of computing processes to the concept of an algorithm, which tends to associate computing with mere calculation without taking concurrent activity into account. Because of the evolution of computing engineering practice, Milner questioned whether the logical grounds of computing should evolve as well. Milner's thesis can be put in a nutshell:

> "this logical foundation has changed a lot since Turing but harks back

> to him. To be more precise: (i) Computing has grown into informatics: - the science of interactive systems; (ii) Thesis: Turing's logical computing machines are matched by a logic of interaction" [184].

Consequently, a theory and a new language to express concurrent activities were required:

> "we must find an elementary model which does for interaction what Turing's logical machines do for computation" [184].

The need to define a new computing theory is thus first displayed through the evolution of computing practice. Milner's motivation and focus were the solving of concurrency issues in distributed systems, with the idea that the evolution of computing practices required new formal tools:

> "Through the 1970s, I became convinced that a theory of concurrency and interaction requires a new conceptual framework, not just a refinement of what we find natural for sequential [algorithmic] computing".

The pi-calculus and his work on the equivalence with automata, known as bisimulation, achieved this reflection on interactive processes [182, 183] with a formalism.

2.1.2 Account for interaction

Milner introduced the opposition between interactional and computational behavior. Using the concept of interaction, Milner referred to concurrent message passing between agents. Milner's work coincided with Petri's new model of concurrent processes [221], intended to describe concurrency in information systems more generally. To Milner, interaction is more expressive than a TM, but it still describes an effective procedure. Milner did not assert equivalence between an interactive model and a TM, but he introduced the topic [183] and he seemed to have left it unanswered.

Four main differences between *old* (computational) and *new* (interactional) computing are made striking by Milner. First, in Milner's words, a Turing Machine prescribes a behavior to be executed. By contrast, new computing requires describing an information flow between several system components. Second, old computing is characterized by a hierarchical design when current practice involves heterarchical phenomena in the computing system. Third, in new computing, the designer cannot predict when agents will be triggered or the overall behavior of the computing system. Fourth, the user is not merely looking for an end result in

new computing practice. There is more than a mathematical function to evaluate, as it used to be in old computing. The user interacts with the system, and the look for an end result is replaced by continuing interaction. Having taken stock of the evolution of computing practice on the engineering side, Milner examines its consequences on the logic foundations of computing. He proposed a new calculus, the pi-calculus or the calculus of communicating systems, to offer a new interactive model [183].

2.1.3 Legacy

Milner's work on interaction has become a founding block in automata theory and concurrency theory. The pi-calculus has inspired research to derive the Pict [229] programming language. His work is foundational and served as a reference for anyone, reflecting on the need for a new framework dedicated to new emerging computing practices. Milner insists on an important reminder that we would like to consider. When modeling, the engineering practice matters and is to be articulated with the logical foundations of the model, possibly involving elaborating a new framework. Famously, Wegner and Goldin acknowledge that Milner was the first to introduce the idea that classical models of computation were insufficient. They argue that Milner did not state clearly whether the computation of CCS and the pi-calculus were irreducible Turing machines and algorithms [290]. If one goes and looks at Milner's Turing Award Speech, it seems true that classical computation translates into an interactive calculus. But it is not stated whether any formula in the pi-calculus can be expressed in a classical calculus like the lambda-calculus. So it is not clear whether the equivalence goes both ways.

2.1.4 Issues for an account of interactive computing

Given the account of current computers that we are looking for, we see two limits in the lessons drawn from Milner.

First, we are looking for an explanation of the interactive computing phenomena at stake in a computer. Therefore, the relation between layers of abstraction, from the computational to the physical, is important. However, to Milner, the physical layer of the machine is not of much interest, and the calculus of communicating systems (CCS) needs to be abstracted away from the physical. As Milner puts it, informatics is about virtual links:

> "physical systems tend to have permanent physical links; they have fixed structure. But most systems in the informatic world are not physical; their links may be virtual or symbolic." [183].

In our perspective, abstracting away from the physical world comes at some cost for an explanation. A good computational explanation should link the formal model and the blueprint of the computing mechanism [187, 190, 192]. Such articulation is not told in a formal theory of concurrent processes. But we will flesh out this argument and tell more about explanations in the next chapter (Chapter 3).

Second, Milner's account of interactive systems restricts them to concurrent systems, which is only one dimension of interest when describing what actual computers do.

2.2 Reactive TMs: extending the original model

2.2.1 Motivation

More recently, a literature domain focused on a "Reactive Turing machine" has developed [9, 14, 15, 152, 159]. The literature reminds us that the purpose of Turing's a-machine was to propose a formal account of what is computable by effective means (algorithmically computable). This formalization was achieved before the realization of the first digital computers. In a way reminiscent of Milner, the question is whether the TM model still fits computing practices decades later. The strategy chosen is to see whether extensions of the original TM are sufficient to describe new computing practices and whether the obtained model is still equivalent to a TM. The strategy founds its frame within computability theory and reflects on its scope. In that respect, although pointing at the specificity of interactional behavior, the main framework still relates to Turing's. Baeten [15] is looking for a computational model of interaction, extending the classical TM with a process-based theoretical notion of interaction related to Milner's previous work.

The strategy involves questioning the relationship between such extensions and the Church-Turing thesis. As a reminder, the Church-Turing thesis states that a computable function by effective means is computable by a Turing machine. The community interested in Reactive Turing machines asks the following question: can the Church-Turing thesis also be extended? Van Leeuwen [151] focuses on the possible extension of the Church-Turing thesis to account for interactive computing:

> "We will motivate the need for a reconsideration of the classical Turing machine paradigm and formulate an extension of the Church-Turing thesis" [151].

What is at stake is whether the Church-Turing thesis holds given required new models of computation:

> "The emphasis in modern computer technology is gradually shifting away from individual machines towards the design of systems of computing elements that interact. New insights in the way physical systems and biological organisms work are uncovering new models of computation. Is the Church-Turing thesis as we know it still applicable to the novel ways in which computers are now used in modern information technology? Will it hold for the emerging computing systems of the future?" [151].

The Church-Turing thesis, it should be noted, does not entail a claim about computing in general (what computers do and will do). Understanding computing from a formal perspective consists of questioning what can be computed and seeing if there is another notion of computation than effective computation in the sense of Church-Turing. In other words, when describing computing within the frame of computability theory, the question about computing is substituted by a question about computation. Once again, what matters to us is whether such a perspective explains interactive computing.

2.2.2 Account for interaction

In the Reactive TM community, the starting point is a standard current computer designed as a distributed system interacting with an environmental agent. They label such a computing system a "site machine" [151]. Starting from this model, the reflection on interaction aims at showing the equivalence between this site machine and a Turing machine augmented by some functions. The conclusion is thus the following: a site machine computer computes effectively and yet requires a model with new functions and thus requires an extension of Church-Turing's thesis. There are effectively computable functions that TMs in the strict sense cannot compute. One crucial dimension that the community wants to account for is particularly relevant to us:

> "In order to mimic site machines, a Turing machine must have a mechanism that will enable it to model the change of hardware or software by an operating agent" [151].

To make interaction with an external agent possible, the model needs to integrate a way of entering new, external, and possibly non-computable information into the machine. This is what oracles do [1]. The authors prefer a more general notion: an

[1] Turing introduced "oracles" in his Ph.D. dissertation [276]. Turing had extended the automatic-machine (what is usually referred to as the "Turing Machine", as described in Tur-

advice function. The model of a Reactive TM (also called a TM with advice) is considered expressive and definitionally equivalent to an oracle Turing machine.

Van Leeuwen identifies three key elements that should be integrated all together within the frame of algorithmic computability: "non-uniformity of programs", "interaction of machines", and "infinity of operation". By the "non-uniformity of programs", Van Leeuwen refers to the fact that current programs on a personal computer are no longer fixed but evolve, are upgraded, and their data remain in memory even when the machine is not running. By "interaction", he intends to contrast a TM where all input data are present before the start of the computing procedure with a modern computer where continuous streaming of data via input ports is going on. The third mentioned characteristic, the infinity of operation, refers to the problem of distributed systems and mobile communicating systems. They are to be seen as dynamic networks of many entities sending and receiving signals in unpredictable ways that are to be synchronized. To accommodate the original TM model, Leeuwen proposes to define "Interactive Turing machines with advice". Integrating an "advice" function amounts to entering new, external, and non-computable information into the machine, which requires the use of oracles [17, 106]. This way, a TM with advice resembles site machines and I/O automata: the TM with advice is equipped with input and output ports. To Leeuwen, formal tools to support interaction and infinite computations are already available. As for interaction, he refers to already well-known and developed literature on the theory of concurrent processes, the programming of parallel processes, communication protocols, and distributed algorithms. As for infinite computations, Leeuwen understands them from the language-theoretic viewpoint in the theory of omega-automata [269, 273].

2.2.3 Legacy

This approach to extending the Turing machine and the Church-Turing thesis is at the junction between Milner's and Wegner's work (presented in the coming section). It makes the junction in that it poses the question of a new paradigm, a question that Milner had not formulated in such radical terms and that is fully defended by

ing's seminal paper [275]). Turing had thought about formalizing the solving of uncomputable problems. In an automatic machine, all data is given before the execution starts, and there is no means to change the symbols on the tape once the execution is launched (the tape header can write and erase symbols — but these are given prior to execution). But an oracle machine can consult an oracle during an execution step, being provided with a new symbol (possibly an uncomputable one) during execution. The halting problem becomes then solvable. Turing's work on oracle machines was expanded later by Post [231]. See, e.g., Soare's work [267, 268] for a more detailed study on the introduction of oracles by Turing and how Post expanded Turing's ideas.

Wegner. The Reactive Turing Machine community begs the question of whether the mentioned required extensions lead to a new computing paradigm:

> "The experience with present-day computing confronts us with phenomena that are not captured in the scenario of classical Turing machines" [151].

The computations carried out on Turing machines with advice are more powerful than classical computations on a-machines. The authors insist that this claim does not go against the Church-Turing thesis. To Leeuwen, like other physical systems [232], TMs with advice or oracle Turing machines do not fit the concept of a finite algorithm that can be computed by means of a TM. The conclusion pushes towards a paradigm shift:

> "What makes them non-fitting under the traditional notion of algorithms is their potentially endless evolution in time. This includes both interaction and non-uniformity aspects. This gives them the necessary infinite non-uniform dimension that boosts their computational power beyond that of standard Turing machines."

The authors ensure that such a paradigm shift does not question the original Church-Turing thesis because their proposal for interactive computation does not involve solving undecidable problems [151] by means of effective computation. However, their work seems to be pivotal in the debate on a model of interactive computing. It has grounded the debate around the implications of an interaction model for the Church-Turing thesis. At least, it can be observed among the objections formulated against Wegner's work. As we will detail in the next section, Wegner pushes further the concept of interaction and the need for a new paradigm, and his proposal falls under objections framed within the computability theory. We will see that the core objections to Wegner ask whether his claim threatens the Church-Turing thesis.

2.2.4 Issues for an account of interactive computing

The project is focused on extending the original TM to make it "reactive". The proposed level of abstraction cannot account for the mechanisms that make the proposed extensions possible. Therefore, we can take a closer look at the type of description proposed in this formal framework to account for an interactive scenario:

> "The computational scenario of an interactive Turing machine is as follows. The machine starts its computation with empty tapes. It is driven by a standard Turing machine program. At each step, the machine reads the symbols appearing at its input ports. At the same time, it writes some symbols to its output ports. Based on the current context, i.e., on the symbols read on the input ports and in the 'window' on its tapes, and on the current state, the machine prints new symbols under its heads, moves its windows by one cell to the left or to the right or leaves them as they are, and enters a new state. Assuming there is a move for every situation (context) encountered by the machine, the machine will operate in this manner forever. Doing so, its memory (i.e., the amount of rewritten tape) can grow beyond any limit. At any time $t > 0$ we will also allow the machine to consult its advice, but only for values of at most t." [151]

If we look for a mechanistic explanation of computing, we need some elements to be unpacked beyond a formal account. We can mention at least two of them. First, we need to account for how reading and writing on the ports are possible. It presupposes that the interactive computing system can react upon arrival of new data. What allows such behavior? Second, in the case of input and output ports, the symbols to be expressed must refer to complex physical phenomena with several dimensions. For example, a user's gesture on a screen, providing an input, carries at least two dimensions: coordinates and pressure level on the interface. A mechanistic explanation needs to account for the richness of external signals interacting with the computing processes. In other words, given the initial question ("what is an interactive computer?"), some phenomena cannot be accounted for within the frame of an extended Turing machine. The way oracles work remains at a level of abstraction too remote from the minimal causal blueprint we need for our purpose.

2.3 Going beyond TMs? Wegner's new paradigm

2.3.1 Motivation

A strong motivation for Wegner's view on interaction is to overcome the Strong Church-Turing thesis (Strong CTT). Wegner thinks the Strong CTT prevents us from fully admitting a new paradigm in computer science. A paper fleshes out in detail clarifications against the Strong CTT [102]:

> "The classical view of computing positions computation as a closed-box transformation of inputs (rational numbers or finite strings) to outputs.

> According to the interactive view of computing, computation is an ongoing interactive process rather than a function-based transformation of an input to an output. Specifically, communication with the outside world happens during the computation, not before or after it. This approach radically changes our understanding of what computation is and how it is modeled. The acceptance of interaction as a new paradigm is hindered by the Strong Church-Turing Thesis, the widespread belief that Turing Machines (TMs) capture all computation, so models of computation more expressive than TMs are impossible." [102]

In other words, the Strong CTT stipulates that a Turing machine could solve all computational problems and could compute anything that any computer can compute. Wegner argues Turing himself would have denied it, referring to Turing's seminal paper [275]. In that paper, Turing not only introduced TMs (calling them automatic machines or a-machines) but also introduced choice machines (c-machines), extending TMs by allowing a human operator to make choices during the computation. Turing did not view c-machines as reducible to TMs, suggesting other forms of computation might exist. Wegner also likes to remind us that CTT applies only to the computation of functions rather than to all computation:

> "Function-based computation transforms a finite input into a finite output in a finite amount of time, in a closed-box fashion. By contrast, the general notion of computation includes arbitrary procedures and processes – which may be open, non-terminating, and involving multiple inputs interleaved with outputs." [102] .

For the sake of clarity, he proposes to formulate explicitly the CTT's assumption in their proper formulation free of extrapolation [102] :

- "All algorithmic problems are function-based."
- "All function-based problems can be described by an algorithm."
- "Algorithms are what early computers used to do."
- "TMs serve as a general model for early computers."
- "TMs can simulate any algorithmic computing device."
- "TMs cannot compute all problems, nor can they do everything that real computers can do."

One reason the Strong CTT is "impossible" is that no computable function would determine, given some finite amount of a priori information, all the real-world factors necessary to complete some task. An assertion to the contrary would endow TMs with the power to predict the future.

The motivation that goes hand in hand with this discussion against the Strong CTT is a reflection on algorithms and the scope of algorithmic problem-solving. Knuth gives a classical definition for algorithms: "An algorithm has zero or more inputs, i.e., quantities which are given to it initially before the algorithm begins." Knuth distinguished algorithms from the arbitrary computations that may involve I/O. One example of a problem that is not algorithmic is the following instruction from a recipe [135]: "toss lightly until the mixture is crumbly." This problem is not algorithmic because a computer cannot know how long to mix; this may depend on external dynamically changing conditions, such as humidity, that cannot be predicted with certainty ahead of time. In the function-based mathematical worldview, all inputs must be specified at the start of the computation, preventing the kind of feedback that would be necessary to determine when it is time to stop mixing. Another example is the problem of driving home:

> "the problem of driving home from work is computable – by a control mechanism, as in a robotic car, that continuously receives video input of the road and actuates the wheel and brakes accordingly. This computation, just as that of operating systems, is interactive, where input and output happen during the computation, not before or after it."

Wegner argues that such a notion of computation does find its counterpart neither in the theory of computation nor in the concurrency theory.

Therefore, Wegner introduced interaction as a new paradigm, based on an empiricist approach [285], to broaden algorithmic problem-solving [103]:

> "Computational problem solving requires open testing of assertions about engineering problems beyond closed-box mathematical function evaluation. We have therefore proposed interactive computing as an empiricist model that expands computational problem solving from algorithmic TM models and functional input-output to broader concepts of interleaved dynamic streams and observable interaction with the environment." [291].

The reason is that he takes computing machines to be about physical processes, chaotic in nature [262], requiring demanding precision to be controlled [116]. A multilayering of abstractions allows us to describe and control those physical and chaotic computing machines. The challenge is then to bridge the gap between all those layers of abstraction, starting with the lowest physical level.

2.3.2 Account for interaction

This leads us to Wegner's account for interaction:

> "We have therefore proposed interactive computing as an empiricist model that expands computational problem solving from algorithmic TM models and functional input-output to broader concepts of interleaved dynamic streams and observable interaction with the environment." [291]

In Wegner's perspective, interactions are more powerful than Turing machines with finite initial inputs. Interactive systems are more accurately modeled by Turing machines with oracles and unbounded (dynamically extensible) input streams than by traditional Turing machines [87]. Oracles were shown by Turing [276] to be capable of computing more than the recursively enumerable functions. Interactive systems react dynamically to external events. They are also related to the passage of external time. By delaying the binding time of inputs so that they can occur during the computation (rather than only at the beginning) and modeling reactive processes [169] by infinite computations [273] the modeled entities are extended from algorithms to persistent objects, and concurrent processes [183].

Wegner wonders whether Milner himself avoided questioning whether the computation in CCS and the pi-calculus went beyond Turing machines and algorithms [290]. The question could remain whether Wegner takes interaction as a super-calculus/super-algorithm or as a radical shift from TMs. In other words, to what extent is "interaction more powerful than algorithm" [286]?

In fact, Wegner's claim is sharp. In contrast with Milner, Wegner's focus is not on concurrency between computing processes. He focuses instead on the complexity of the triggering of external events outside the machine:

> "Interactive systems are grounded in an external reality both more demanding and richer in behaviour than the rule-based world of non-interactive algorithm." [286]?

He strikes the difference between closed and opened systems, the latter being impossibly completely described. This impossibility makes interactive systems mathematically problematic: they lack completeness.

> "The comfortable completeness and predictability of algorithms is inherently inadequate in modelling interactive computing tasks and physical systems. The sacrifice of completeness is frightening to theorists who work with formal models like Turing machines (...). But incomplete

> behaviour is comfortably familiar to physicists and empirical model builders. Incompleteness is the essential ingredient distinguishing interactive from algorithmic models of computing and empirical from rationalist models of the physical world."

From this, Wegner concludes that computing systems should not be thought of as algorithms but as interfaces, views, and modes of use, definable as behaviors to be specified. Consequently, an ontological question is also at stake: in what terms should the external world be modeled as atomic objects and events? as processes and flows?

Formally, Wegner's account of interaction has led to the development of Persistent Turing machines (PTMs), a model of sequential computation, and to the result that multi-stream interaction machines (MIMs) are more expressive than sequential interaction (SIMs). Wegner and Goldin trace back the idea that interaction is not expressible by or reducible to algorithms to the closing conference on the 5th-generation computing project in the context of logic programming in 1992: reactiveness of logic programs, realized by the commitment to a course of action, was shown to be incompatible with logical completeness [289].

2.3.3 Legacy

The main objection made by other researchers to Wegner's work is that interaction machines can be proved equivalent to TMs. The objections are focused on the defense of the Church-Turing thesis [64, 233], assuming that interactive modeling is a way of denying the results of Church's and Turing's work. However, this assumption cannot be taken for granted: no one denies that TMs and the lambda-calculus account for effective computation. Both formalisms define the intuitive notion of an algorithm. The Church-Turing thesis will remain unshaken until someone presents an alternative formal account of an effective procedure. Due to semantic ambiguities, some have interpreted Wegner's work as challenging the Church-Turing thesis. First, Wegner characterizes interaction as more powerful than algorithms and TMs. What "powerfulness" precisely refers to is unclear. We will say more about this in the next chapter (Chapter 3, Section 3.1). Second, there seems to be another semantic ambiguity or alleged identity between "computing" and "computation": "Wegner (and Eberbach) say that it is impossible to describe all computations by algorithms. Thus, they do not accept the classical equation of algorithm and effective computation" [64]. In the former quoted sentence, a core assumption uses interchangeably "computation" and "computing". But Wegner means that it is impossible to describe everything in computing by algorithms. By *computations*, he is referring to what computers can do in a broad sense, not to

	Interaction as concurrent communicating systems	Interaction as extended Turing Machines	Interaction as a new paradigm
Motivation	Provides new logical foundations to fit new engineering challenges, especially concurrency	Extends the TM model to account for interactive devices	Debunks the strong Church-Turing thesis Discusses the scope of algorithmic solving Prones the need for a new computing paradigm
Account for interaction	Information flow Heterarchical design No complete prediction about overall behavior No end-result Process calculi	External data needed during computation Non-uniformity of programs Interaction with agents Infinity of operations Interactive machines are TMs with advice	Computers have rich interaction with the environment during computing execution, but this processing is not merely algorithmic
Uses and criticisms	First conceptualization of interaction Legacy for automata theory	Inspires the need for a new paradigm Puts at the forefront the Church-Turing thesis	Controversy about the powerfulness of the TM
Issues for an account of interactive computing	Definition of interaction restricted to specific properties: concurrency and communication	Formal oracles cannot account for the physical possibility of entering new data	Issues about powerfulness and expressiveness constrict the debate in the realm of computability theory

Table 2.1: Sum-up: an overview of explicit theories of interactive computing systems in theoretical computer science

effective computation in a narrow sense. Therefore, the conclusion made in the quoted sentence does not follow: the identity between an effective computation and an algorithm is not put into question by Wegner. Wegner does not claim that his view on computation can solve the halting problem.

2.3.4 Issues for an account of interactive computing

We are interested in the way Wegner broadens the notion of interaction. It is not strictly referring to communicating processes within a machine. Possible complex interactions with the environment are considered. But although debunking the focus of the CTT by stating that interaction is more powerful and expressive than algorithms, Wegner's work is enclosed in a field of discussion framed by computability theory.

Furthermore, we still lack a way of describing the very mechanisms we are interested in to be provided with a mechanistic account of interactive computing systems. And this is no surprise since Wegner's work aims primarily to reflect on the theoretical limits of classical mathematical tools, e.g., the notion of completeness.

2.4 Summary

We have reviewed the conceptualization of interactive systems in theoretical computer science. We argue that these approaches cannot answer our epistemological question about the characterization of interactive computing. There are two reasons for this. First, there is an unclear stance towards interaction models and their reducibility to classical models of computation. Second, as we have seen, these conceptualizations focus on whether a formal model for interaction is reducible to a Turing machine and, if so, whether this is a threat to the Church-Turing thesis. This deprives us of a level of description that would explain the mechanisms allowing a computing system to be interactive.

The first problem with the focus on Turing reducibility in the account for interaction is that the stance is not clear-cut. Milner's work leaves us with the following question: to what extent are the new "logical foundations" for interaction distinct from the classical framework? Irreducibility is not stated in the speech for the Turing Award. There is a simple translation of lambda-calculus into pi-calculus, which is faithful to computational behavior. Thus, pi-calculus supports functional programming at a higher level of explanation. But it is unclear whether any behavior expressed in the pi-calculus can be translated into a classical calculus. In a more recent book, *The Space and Motion of communicating agents*[185], Milner introduces bigraphs as another formalism for interactive systems. Bigraphs are proved to have the same expressiveness as Turing machines. It looks like Milner proposes to revise the principle of Occam's razor and praise the plurality of formalisms, models, and frames of explanation:

> "I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent computation, which is in a sense the whole of our subject — containing sequential computing as a well-behaved special area. We need many levels of explanation: many different languages, calculi, and theories for the different specialisms." [182]

It looks like interaction is the new "basic notion":

> "Now, what are the new particles, parts of speech, or elements which allow one to express interaction? They lie at the same elementary level as the operation of a Turing machine on its tape, but they differ. For much longer than the reign of modern computers, the basic idiom of algorithm has been the asymmetric, hierarchical notion of operator acting on operand. But this does not suffice to express interaction

> between agents as peers; worse, it locks the mind away from the proper mode of thought." [184]

Is Milner's proposal still equivalent to Turing Machines, or does it go beyond? Does it mean interaction is something else, something irreducible to TMs? Does interaction amount to old computing with extended computational power? The same question arises regarding the claim made by the Reactive Turing Machine or by Wegner.

The main problem is that theoretical computer science frames the question about interaction in terms of extensions of the TM or the reducibility of models for interaction to the TM. It ends up reducing the question to a debate about the Church-Turing thesis: do interactive computing systems carry out super-Turing computation/hyper computation? The debate starts with a questionable premise: *that anything a computer does* is a computation. The idea behind this premise is known as the *Strong* Church-Turing thesis. The original Church-Turing thesis has only a claim about effective procedures and their formalizations. A corrected premise would be: computers do computations *but they also do other things*. From that corrected premise, it follows that we need an account and a model for what those other things are. It also follows that discussing interactive computing can be done without committing to any stance towards the nature of computation and the Church-Turing's thesis: the definition of algorithmic computation or effective procedure remains unshaken and is out of the discussion's scope.

Criticisms against Wegner show that the criterion of powerfulness is ambiguous when evaluating a model for a computing system. Does powerfulness refer to computational power, involving that an interactive model can express uncomputable functions in Turing's sense? Or does it refer to the expression of *more phenomena*? In the next chapter, we get into more details on this topic. When offering models for interactive computing and comparing them with classical models of computation, arguments about the powerfulness and equivalence of the models systematically arise.

Chapter 3

Formal models of interactive computation vs. explanations

The literature review leaves us with an unsettled debate about model reduction. Milner, the Reactive Turing Machine community, and Wegner have focused respectively on a set of features to define interactive computing: concurrency, arrival of data from I/O ports, and non-algorithmic problem-solving. Nevertheless, it is left unclear whether these features must be accounted for in a new framework or take place within the classical computability framework. In this chapter, we argue that the debate about Turing reducibility misses the explananda [1] for two reasons.

First, some ambiguities about *model powerfulness* structure the debate and should be clarified. The idea that interaction is *"more powerful than algorithms"*, as claimed by Wegner, can be interpreted against the Church-Turing thesis. In that case, the polemic arises because such a claim can only be speculative, pointing in the direction of super-Turing machines computing uncomputable functions. However, there is another possible interpretation of Wegner's work without the commitment to a mathematical revision of the Church-Turing thesis. There could just be *something else* than algorithms to interaction.

Second, focusing the definition of interactive computing around Turing reducibility is a formal debate and does not support an explanation of the relevant phenomena that make interactive computing possible. To reflect on that issue, we introduce a known distinction in philosophy between *formal models of computation* and *mechanistic explanation* [189, 224].

We propose to identify the level of abstraction needed to *explain* interaction and call it an *execution model*. Regarding the level of abstraction, what we call an execution

[1] *Explananda*: what needs to be explained.

model for interaction aims to be the counterpart of the TM for computation. An execution model belongs to a level of abstraction where it is possible to describe a functional architecture and, upon it, in mechanistic terms, an execution. We argue that the TM used to be an adequate execution model for thinking of computing systems but that we need to update it to provide philosophers with a mechanistic explanation of interactive computing systems. As previously fleshed out, since the early days of computability theory, the Turing Machine has been the cornerstone in understanding computers. The TM defines what can be computed and how computation can be carried out. Part of the explanatory power of the TM relies on what we call an *execution model* [2].

We argue that the explanatory power of TM does not embrace interactive systems and leaves a gap. Therefore, there is a need for a mechanistic explanation for interactive computing. The reason is that the fundamental phenomena relevant to interactive computing are out of the scope of classical computability theory. Furthermore, we show that such a model is not available within models of interactive computation [9, 15, 267] either.

3.1 Equivalence and powerfulness of models

As we have seen in the previous chapter (Chapter2), debates about an explicit theory of interaction systematically focus on the *equivalence* between the TM model (or any abstract machine derived from it) and an interactive model. Ambiguities around the concepts of powerfulness and expressiveness likely make the debate confused. Indeed, there are at least two ways of understanding them. In any case, the powerfulness of a model refers to its expressiveness, which is a semantic property.

Expressiveness refers to *what can be expressed* by a given model. If one thinks of a model as a formal language, let us say that expressiveness relates to all the possible sentences one can make in that language. Two structures are definitionally equivalent if and only if they share the same domain and each is definable in the other [47].

[2] Computer scientists are familiar with the term "execution model" that is, for example, used in the field of operational semantics [280, 295]. We use the concept in this Chapter without specific reference to programming languages and their semantics, as simply referring to an intermediate-level description of computing execution. The relationship between an execution model and a programming language is summed-up in the following excerpt from the 1968 Algol report: "The meaning of a program in the strict language is explained in terms of a hypothetical computer which performs the set of actions that constitute the elaboration of that program." ([280], Section 2). However, Chapter 4 will go a bit further and examine the connections between an execution model for interaction and interaction-oriented languages.

In a first sense, powerfulness and expressiveness can be understood strictly within computability theory. In that case, the two notions are used when evaluating a mathematical framework supporting the formalization of semantics. What is called "powerfulness" refers to computational power, and expressiveness refers to a formal criterion evaluating which functions can be expressed. Turing completeness is then a possible evaluation criterion for expressiveness, for instance.

In a second sense, one can consider a the powerfulness and expressiveness of a model *outside the strictly formal computability framework*. Since a model must represent, according to specific objectives, a phenomenon of reality or, say, a system, we can understand the powerfulness of a model as a good match between the model and what is modeled.

Therefore, in that broader sense, a model is expressive, given some purpose, if and only if it describes all phenomena required for some given purpose. In that case, the value of the model and concerns about its expressiveness depend on the stated goals.

From an engineering perspective, for example, a model is valuable to the extent that it allows engineers to think of future systems design easily. In this case, the model's value could be evaluated, e.g., in terms of usability (effectiveness, efficiency, and satisfaction [124]). From a scientific perspective, the aim is to make good predictions about a system. The two perspectives are rarely used in isolation since good engineering design requires some science, and good science often relies today on some engineering [145]. From the perspective of the philosophy of science and given scientific explanation standards, a good model for a phenomenon rightly describes the mechanisms at stake [99, 161, 190]. Of course, other possible values for models, from other perspectives, could be found.

Let us say that among the things that could be expressed in a model are functions and other things than functions.

Within each set, some sets include more than others. The set of hypercomputations within the set of functions is more expressive than the set of computable functions since it includes the uncomputable ones. That is a way to be more expressive: expressing more functions. However, suppose a model allows referring to elements among both sets. In that case, it is more expressive than a model allowing only expressing functions since it expresses more kinds of phenomena (among which are computable functions).

Once again, framing expressiveness and powerfulness as possibly only about computable functions would seem odd to engineers and computer scientists familiar with other formalisms. In a foundational paper, MacCarthy [175] referred to computability theory as one among other strategies to think of programs. He pointed

out why computability theory was not the most fitting theory [3]. Nevertheless, objections about interaction theories frame the debate in reference to computability theory.

To go back to Wegner [104, 285, 286, 287], we argue that this distinction between a narrow and broad sense of expressiveness clarifies criticisms made against him.

In a broad sense, one can interpret Wegner's new paradigm as follows: Wegner considers his interactive model more expressive than a TM by having his model describe *other things than functions*. Wegner's model could then describe more phenomena than a TM. The interactive model would not share the same structure as a TM: the interactive model would include extra elements. It only adds some other represented phenomena (e.g., references to physical time, richer environmental inputs arriving during execution). It would not go against the Church-Turing thesis, which remains valid to account for algorithmic problem-solving through effective procedures.

But in a narrow sense of expressiveness, one can interpret the possibility of a new paradigm as follows. Wegner and the tenants of Reactive Turing Machines could think of their interactive model as more expressive than a TM, allowing their model to execute *more functions*, even some of them being uncomputable functions in the sense of the Church-Turing thesis. In that case, the claim would indeed be controversial. The bold claim would be the following: a TM is not only providing an account for algorithmic problem-solving through effective procedures but it could also be extended to account for other non-algorithmic processes, solving the uncomputable. Interaction would be some super-calculus, extending the calculative power of the original TM to account for interaction. It would be satisfactorily modeled with a TM, only given more calculation power. It goes down the track of Accelerating Machines or Super-Turing Machines, able to calculate more than Turing's computable functions [70, 71, 163].

We argue that a theory of interaction does not need to embrace the hypercomputation view. Part of an interaction model could be reduced to the classical TM, but some extra elements needed to express interaction cannot be reduced to an a-machine. That does not mean interactive models have super computational power to solve undecidable problems. It simply means interactive systems do things that

[3] To quote McCarthy [175], pp.1-2: "There are three established directions of mathematical research relevant to a science of computation. (...) The second relevant direction of research is the theory of computability as a branch of recursive function theory. The results of the basic work in this theory, including the existence of universal machines and the existence of unsolvable problems, have established a framework in which any theory of computation must fit. Unfortunately, the general trend of research in this area has been to establish more and better unsolvability theorems, and there has been very little attention paid to positive results and none to establish the properties of the kinds of algorithms that are actually used. Perhaps for this reason, the formalisms for describing algorithms are too cumbersome to be used to describe actual algorithms".

a TM cannot do. It is possible to admit they do other things *without implying they compute uncomputable functions.*

Our point is the following: it could turn out that equivalence between a TM and phenomena relevant to the design of an interactive device does apply only partially. There are indeed computational procedures among the computing processes of an interactive device. However, this still leaves aside numerous phenomena that are not accounted for in Turing's original model.

Therefore, from our perspective, the equivalence debate is a non-starter. Questioning the possible reduction of a model B to a model A (and therefore the equivalence of A and B), requires that:

- a modeler describes the same types of phenomena, or: any phenomenon of interest in B must be accounted for in A.
- a modeler has the same purpose.

It would amount to a *fallacy of composition* to assume any equivalence between A and B, due to an equivalence between A and **a part of** B (or vice versa).

This is obvious to engineers who have needed to develop new formalisms to account for new properties without reasoning in terms of Turing-reducibility. However, in the more restricted field of theories for interaction, this framing with reference to classical computability is still at stake.

3.2 Formal models and mechanistic models

To understand what makes the specificity of interactive computers, let us go back to our simple introductory example: a drawing application on a smartphone (Figure 1). As simple as it is, this example reveals some interesting phenomena that we detailed previously in the introduction.

None of these phenomena can be explained in the classical epistemic framework, where a computer is understood on a formal level through computability theory and on a concrete level through a specific model of computer architecture. On the formal side, we are left with automata theory and formal language recognition. On the concrete side, we are provided with a specific computer architecture model and its functioning: a binary alphabet, a memory unit, and a processing unit whose execution stops when the output has been reached. When trying to describe current interactive computers, we argue that some relevant phenomena for an explanation are simply out of the scope of the classical framework.

Faced with this discrepancy between the classical framework and the actual functioning of computers, some philosophers have encouraged a philosophical reevaluation of the epistemological stance toward computers. Philosophical accounts have delineated instead typologies to distinguish types of computers in terms of functionalities [225] or have deconstructed the idea that computability theory is *per se* a complete account of computing phenomena [266]. Others have developed the idea that transcending Turing computability is required to deepen our understanding of computation for mathematical [82, 102] and epistemological [162] reasons. The debate asks whether the classical notion of computation can capture relevant phenomena to formalize (mathematically) and understand (epistemically) current computing systems. However, to the best of our knowledge, no mechanistic description of interactive computing has been proposed.

To get a clearer epistemic view of interactive computers, we propose to introduce our notion of an *execution model* [4]. What we call an execution model explains what a computer does by describing in mechanistic terms how instruction is carried out on some functional architecture. It is posited at a level of abstraction between the purely formal model of computation and the physical-mechanical model of the computer's hardware.

Computing in the early days could find its execution model in the Turing Machine. However, the TM was not initially concerned with modeling an actual computer (computers did not even exist in 1936). But the singularity of the TM and its explanatory power relied at least upon its ability to be a twofold abstraction. Not only could the TM support the formalization of the set of computable functions (what can be computed), but it also provided the intuition of the mechanism [39] that could achieve such a computation (how it can be executed). This specific feature had already been commented on by Gödel, when comparing the TM to other equivalent formalisms [259]. To refer to the twofold nature of the TM, we use in this chapter a distinction between a *model of computation* and an *execution model*. On the contrary, other models of computation, such as lambda-calculus, do not carry any reference to execution and do not allow any mechanistic description of a computational behavior.

In this chapter, we argue that the execution model provided by the Turing machine in the early days of computing cannot account for the properties of interactive computing systems today and that there is currently no candidate among interactive models of computation for an updated execution model. We are therefore left with

[4] The concept of an *execution model* is already available in operational semantics, but we are using it in this thesis without direct commitment or reference to operational semantics, although our notion and the original one share the same interest: the description of an execution. The difference is that we want to use the concept in broader sense as an epistemological tool, and not in a more restricted use within the frame of operational semantics.

an explanatory gap. One needs to provide an adequate execution model to have an explanatory account of the existing computing systems. Such an execution model does not need a reference to calculations but needs to make sense of the causal orchestration between physical processes.

The rest of the chapter is organized as follows. We present in Section 3.2.1 the historical and theoretical grounds for understanding digital computers by referring to Universal Turing Machines. Section 3.2.2 introduces the evolution of computing practices and computing modeling in more detail, fleshing out the concept of interactive computers. It shows why the Universal Turing Machine and new formalisms cannot explain interactive computing phenomena mechanistically. Section 3.2.3 conceptualizes the difference between models of computation and execution models. Section 3.2.4 posits our concept of execution model among other approaches to computational explanations.

3.2.1 Digital computers, computing mechanisms and computability theory

The epistemology of computers has relied on a reference to the Universal Turing Machine (UTM) [5] for historical and conceptual reasons. It is not surprising since the UTM is not only a model of computation but also describes a mechanism and suggests that a minimal set of components suffices for a functional architecture. In this section, we briefly remind the connection between computability theory and the epistemology of computers. We question whether that connection still supports a mechanistic description of computing behavior.

3.2.1.1 The Universal Turing Machine (UTM): a model for the digital computer?

Mechanistic explanations have become the standard in science to account for explananda phenomena. Digital computers are no exception, and the straightforward way to address what they are is to provide a mechanistic description of the relevant computing mechanisms at stake. There are many levels of abstraction to describe what a digital computer is. For example, one could focus exclusively on the description of what is going on in terms of voltage firing from the point of view of an electronic engineer. Alternatively, one could focus on the transformations in memory registers or describe the program running on the machine. The complexity of every layer makes it difficult to articulate an overall explanation of how the system works, paying tribute to every single layer (see Figure 3.1).

[5] A Universal Turing Machine is a special Turing Machine that can simulate any other Turing Machine, hence its name.

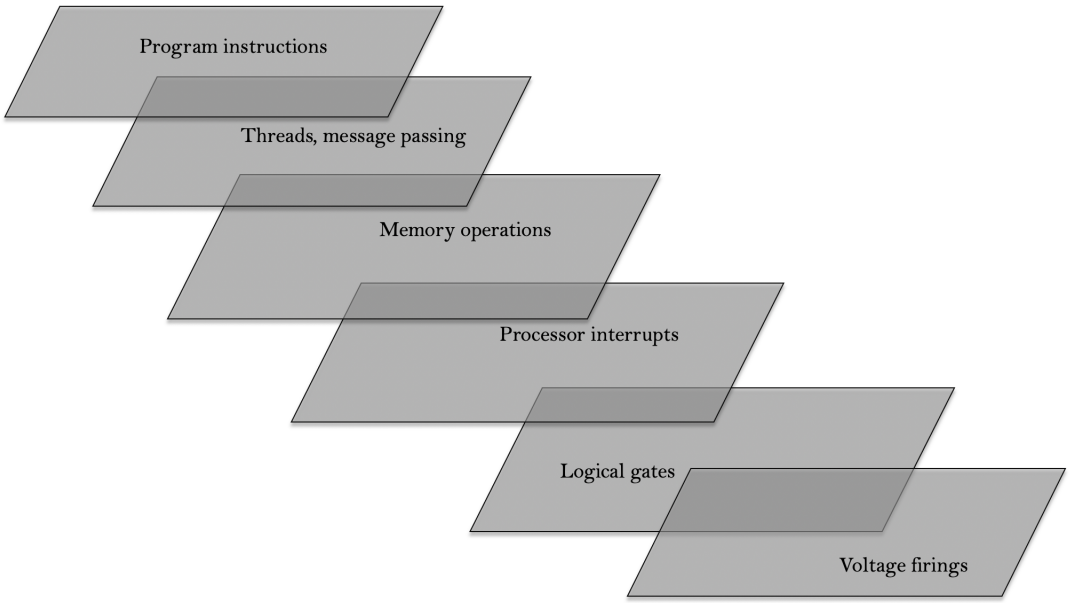

Figure 3.1: Building an explanation for digital computing phenomena: where is the right explanatory focus?

Therefore, from an epistemic point of view, understanding computers is challenging: what is the proper explanatory focus? It is a well-known issue for any mechanistic explanation: "it is sometimes possible to decompose a system at high or too low a level and miss the level at which interactions transpire that are crucial to accounting for the phenomenon in question." [24]. Engineers can flesh out how each layer functions and how the layers are related to each other [147, 216]. However, they admit it is impossible to provide a detailed overview of a computing system within a single abstraction. Nevertheless, this does not mean that epistemologists must give up searching for an explanatory abstraction. It only shows that any explanatory abstraction for a computing system is necessarily a trade-off: it cannot be both exhaustive in detail and understandable by a human.

Traditionally, philosophers have been describing the computing mechanisms in a computer in terms of the manipulation of digits [223, 226]. In that case, a computing system is mechanistically described through its capacity to generate output strings of digits from input strings of digits and (possibly) internal states following a general rule.

Such an approach finds tools and models for a mechanistic explanation within the frame of computability theory. State automata, and among them the Universal Turing Machine (UTM), match the targeted level of explanation: it is the very job of a state automaton to model the transitions from one state to another, following rules step-by-step. Therefore, it is not surprising that the epistemology of digital

computers, computing mechanisms, and computability theory have become related to each other in the philosophy of computing. That situation is also a reminder of the peculiar status of the Turing machine in computability theory, among other models of computation. Contrary to lambda calculus [6], the Turing machine does not only formalize the notion of an algorithm or make proof about computable numbers. It also carries a hint about a computing mechanism.

In other words, it is a two-faced abstraction: it supports both the description of a mechanism and a mathematical proof. Therefore, philosophers have tended to focus on the Turing machine among all models of computation.

3.2.1.2 Historical perspective

The Turing machine has not only been used as a building block of computability theory. It has also been viewed as the model of the modern computer [193]. The usual explanation for the latter relies on the similarities between the blueprint of the first modern computer as historically found in von Neumann's EDVAC design with its stored program and the architecture of classical computers (having a von Neumann architecture). Such an architecture involves storing instructions and data in the same memory. Although discussed [75, 111, 112, 113, 118, 199], it is sometimes alleged that von Neumann had read Turing's paper [74] and had derived the stored-program concept from the Universal Turing Machine (UTM). Because the UTM provided an abstraction to think of a stored program, reasons were found to make the UTM more than an abstraction for computability theory and make it the right abstraction to describe the modern computer. Another argument favoring the UTM as the origin of the physical computer is Turing's work on actual computing devices during World War II (the Automatic Computing Engine (ACE)). It could suggest that Turing had already thought of the physical computer when introducing the UTM.

Historical deepening of the question would nuance the idea that the Turing machine supported the invention of modern computers, although it was a major contribution. However, the UTM became very influential to think of computer architecture:

> "In the 1950s then the (universal) Turing machine starts to become an accepted model in relation to actual computers and is used as a tool to reflect on the limits and potentials of general-purpose computers by both engineers, mathematicians and logicians. More particularly, with respect to machine designs, it was the insight that only a few number of operations were required to built a general-purpose machine which

[6] We will make later on a caveat in that respect, mentioning the exception that is the implementation of the Lisp machine.

> inspired in the 1950s reflections on minimal machine architectures." [193]

Today, the UTM is still considered in philosophy as the modern computer model. However, computer scientists have already argued that such a view is mistaken [146, 162]: "(...) computers can do things that a universal Turing machine cannot. Many applications, including Wikipedia and Google search, are designed never to terminate and are interactive" ([146], chapter 8). Of course, other models have been introduced since to account for the new properties of digital computers. However, as we will see in the coming sections, they are mere models of computation and are not execution models.

3.2.1.3 Digital computers and interaction: is the UTM a definitive model?

Since the UTM has been used as a model to understand what computers are, computing practices have evolved. A legitimate question is then whether the UTM pays justice to current practices. As said before, computers are increasingly interactive. They are no more transformational systems producing a final output after a finite execution. Instead, they continuously react in time to external events that modify the course of execution. In fact, the concept of interactive computing has been theorized at least since Milner [182] to account for the new properties of computing systems. Let us have an overview of these new properties.

First, computer scientists wanted to formalize a new kind of execution that does not terminate and can integrate new incoming data during its course. In an old-fashioned UTM, all data is given before the execution starts and there is no means to change the symbols on the tape once the execution is launched (the tape header can write and erase symbols — but these are given prior to execution). Turing had already thought about formalizing the arrival of external input data provided during an ongoing execution: he introduced such an abstraction in his Ph.D. dissertation [276] and coined it the "oracle machine". The oracle machine has nothing to do with interactivity, it is originally intended to serve as an infinite table lookup, supporting the solving of undecidable problems. Turing's idea was expanded later by Post [231] [7]. Turing's and Post's work has later inspired derived formalism on "extended" Turing Machines, such as the "Reactive Turing Machine" [9, 15, 159] or the "Persistent Turing Machines" [104], that we detailed in Chapter 2.

Second, since the 60s, computer scientists have had to address the modeling of concurrency in distributed computing systems, where more than one thing could

[7] See, e.g., Soare's work [267, 268] for a more detailed study on the introduction of oracles by Turing and how Post expanded Turing's ideas.

happen at once between several computing components. In an old-fashioned UTM, it is the step-by-step execution of a given procedure that is described. It is not about the synchronization of message passing. This led to the first labeling of "interactive" computing in the 80s by Milner, who theorized that computing systems had evolved into communicating systems, where the synchronization between messages was of more importance than computation to build well-functioning distributed systems [182, 183, 184].

Third, time has become a concern for a formalization [7, 144]. In an old-fashioned UTM, time is abstracted away or reduced to a logical notion: there is only a notion of sequenced steps but no reference to quantified physical time. For example, there is no specification of the time it takes to move from one tape cell to another. On the contrary, to program human-computer interactions, it is important that a computer nowadays follows precisely timing constraints. In HCI, programmers are often confronted to specification of timing and delays, e.g. when coding an animations, or a double click. What makes the measurement of physical time even more crucial, is the necessity to adapt the computing system to human perception [50, 213]. This was an issue commented at length by some participants (see Chapter 1) who also introduced the notion of "haptic bugs" (when an animation does not behave correctly e.g. too quickly or slowly, according to the human eye). A glimpse at the literature in computability theory shows that new models of computation addressing incoming data during ongoing execution, concurrency, and time issues, systematically put at the forefront whether they are equivalent to UTMs. It is important to notice that the answer to such a question does not tell us whether these new models of computation are good mechanistic descriptions of current interactive computing systems. It is thus left to the investigation: are the new models of computation relevant to descriptions of computing mechanisms, supporting a mechanistic explanation of interactive computers? We argue in the following section that they do not.

3.2.2 What models of computation cannot explain: digging into details

There has been plenty of work on the formalization of interactive computing properties. Formalisms have targeted the representation of (i) the arrival of external input data and ongoing execution, (ii) concurrency and synchronization between processes, and (iii) time. We want to review these formalisms and detail why they lack what we are looking for: an execution model for a mechanistic account of interactive computing behavior. We argue that existing (interactive) computation models do not provide the minimal requirements and necessary ingredients for an execution model. We will delineate why the formal approach does not give us a

mechanistic account in each case.

3.2.2.1 Formalizing arrival of external input data and ongoing execution: oracles and extended Turing Machines

Historically, the first formalized aspect of interactive computing was the arrival of external input data provided during an ongoing execution. If one imagines a Turing Machine, this amounts to modeling the possibility of some external writing of new symbols on the tape during the execution. As we previously said, Turing introduced such an abstraction in his Ph.D. dissertation [276] and coined it the "oracle machine". His work was expanded later by Post [231]. Turing's and Post's work has later inspired derived formalism on "extended" Turing Machines, such as the "Reactive Turing Machine" [9, 15, 159] or the "Persistent Turing Machines" [104]. Such extensions of the original Turing Machine (the "a-machine") formally account for a necessary feature of an interactive computing device. However, they do not hint at the very mechanism that makes such an interactive execution possible. At least two ingredients are missing here.

First, we are missing the mechanistic description of how such an external action on the tape is made possible. Models of interactive computation with oracles cannot explain it: their job is not to explain how the oracle interacts with the tape: "they cannot be asked to justify the causal/physical chain of their steps" [39]. Data arrival would be useless if it could not trigger something within the system. The specification of an interactive behavior is the description of the relation between the occurrence of an event and the starting of a computational process. An execution model must thus provide the basic mechanisms to control the lifecycle of computational processes from the occurrence of events. Note that this feature is absent from the TM model, as there is no way of describing the machine's launch in reaction to events. To sum up, there is an inversion of control: the original TM entirely controls the flow of steps, while the flow of steps of an interaction machine should be controlled. As we will see in the next chapter, such mechanisms can find different implementation in current computing systems, e.g. interrupts or polling.

Second, the external inputs arriving during execution are often information about external physical processes. An additional mechanism is necessary to translate a physical magnitude into digitalized data. This operation is called transduction. The importance of transducers has already been mentioned in theories of interactive computing [152, 162]. By modeling input as a sequence of symbols, the TM or any extended version does not give insight into the variety of physical phenomena that can trigger computational processes. Some of them can be described as simple Boolean values, a signal being present or not (e.g., mouse clicks). Some others may have a complex structure, such as the continuous change of a physical magnitude

(e.g., light or temperature). The very goal of the TM is to provide an answer to a computation, e.g., whether a number is computable or not. Therefore, there are no dimensions of the inputs and outputs.

3.2.2.2 Formalizing concurrency and synchronization: process calculi, nets, networks

With a growing interest in interactive computing systems, other properties became crucial: the interaction of numerous computing processes running in parallel and communicating. Concurrency theory emerged from Dick Karp's early work in the 1960s, grew with Petri's work [221] and has now developed into a mature theory of reactive systems with diverse network models (for an overview, see [148, 149]). "Interaction" refers to concurrent message passing between agents in distributed systems. In that context, the conceptualization of interactive systems, as opposed to computational systems, emerged in computer science, as formulated by Milner [182, 183, 184].

First, it installed the notion of a transition system as the prime mathematical model to represent discrete behavior [12, 18, 98, 215]. Second, it showed that language equivalence was not the correct notion when comparing automata for interactive systems; instead, it should be replaced by a notion of behavioral equivalence or bisimilarity [183]. Third, it yielded many algebraic process calculi facilitating the formal specification and verification of reactive systems. Those formalisms model message passing but do not target an account about the mechanisms supporting execution. Even without specific mechanisms, synchronization between computing processes is left unexplained. How can the computing processes "wait" for each other and respect an ordering ("if message B arrives before C, then message A should be sent, but otherwise shouldn't")? There is a need, e.g., to launch, pause and resume computing processes upon arrival of messages. Without fleshing out fine-grained details at the processor level, the question remains: how can the system behave the way it does?

3.2.2.3 Formalizing time: temporal logic, timed automata, synchronous models

Time is not a concern in classical automata theory. The Turing Machine or any derived abstract machine specifies how to go from the input to the output in a finite number of steps. But the time it takes to move from one step to another is of no concern. At most, one may be interested in the number of steps from the input to the output. When specifying interactive behavior, a richer notion of time is

required: time as duration or physical time, measured in a physical unit. Duration is essential for several reasons.

First, many interactive systems interact with humans, and human perception is also sensitive to duration. For example, a written message must be displayed for a minimum duration to be readable by a human. Thus, once again, it is necessary to be able to specify a duration, in this case, before stopping a process. The problem with duration is that it cannot be specified by a reference to steps since we do not know the duration of a single step. The only way to specify a duration is by a reference to a physical process whose duration is known, such as the period of a crystal oscillator. Some reference to a physical clock is thus needed to explain how a timing constraint can take place in a computer [8].

Computing devices that interact in time with external events exist, and plenty of solid formalization work about them has supported their design and verification. Timing constraints on real-time systems have notoriously posed a challenge, which has been addressed and solved. Diverse timed automata have served as formalization tools [7, 8, 160, 239, 257]. Nevertheless, a timed automaton does not support our epistemological task. A timed automaton abstracts away from the mechanism at stake and leaves us with time reduced to timing conditions on transitions between states. The reference to a physical clock is out of scope. The mechanism is then left incomplete. Once again, such a reference to a physical device is unnecessary for the mathematician writing proofs about a real-time system. However, it matters to us to figure out what is going on. Furthermore, a timed automaton *per se* is not a model of any physical phenomenon. It has a formal role. For example, it is a way to describe a class of language recognizable by a type of automaton. It would be a mistake to argue that because a timed automaton is a formal model reducible to a TM, the TM expresses time. Such an assertion would be incorrect. The correct rephrasing would be the following: the kind of language the timed automaton recognizes is also recognizable by a Turing Machine. It does not mean that timed automata have provided us with an explanation of how timing between processes occurs in a computing system.

3.2.3 Execution models vs. models of computation

Computability theory has provided a framework for building models of computation. As has been proved at length, different formalizations of the concept of computation turn out to be equivalent, the most famous being Church's lambda

[8] The presence of clock allowing measurement of physical time is distinct from the role of a global clock driving the execution on a synchronous logic circuit. There are systems running on a clockless architecture, but they would not fit the programming of fine-grained interactions such as animations.

calculus [63] and the TM [275]. If formally equivalent, there is a core distinction between the lambda calculus and the TM. The latter does also provide an execution model. We argue that the distinction between a model of interactive computation and a model of interactive execution is also relevant.

It is worth introducing a nuance here, namely that the lambda-calculus inspired later the Lisp language. One of McCarthy's student, Steve Russell (also programmer of the first video game SpaceWar), showed in 1958 that Lisp could serve as a concrete abstract machine and be directly implemented [9] [107].

In any case, Lisp as well as the Turing Machine are the kinds of abstraction that are able to describe an execution.

3.2.3.1 The current purposes of models of computation

Current models of computation in computer science play today a different role than in the 1930s mathematical realm. In computer science, what makes a model of computation valuable is related to the formal properties it expresses. Once those formal properties are at hand, they allow further procedures to be acted upon them, especially system verification and certification.

In the end, models of computation are tools to support and verify a system's design. These models belong to a particular abstraction level: they do not intend to model the system as a whole and the way it works. They focus on verifiable properties, upon which proofs that guarantee the system's outputs are built. We can flesh out an example. Let us consider the design of commands for an airplane. The designer needs to write a program that describes these commands' behavior. It is up to the designer to decide what properties are the most relevant and must be expressed in the model. Those properties to be checked can be, e.g., bounded values for the range of inputs the system can take (to guarantee the "flight envelope", maintaining the correct speed to avoid stall), the absence of infinite loops, and memory overflows. The rest can be abstracted as irrelevant to the specific verification task.

[9] "But in late 1958, Steve Russell, one of McCarthy's grad students, looked at this definition of *eval* and realized that if he translated it into machine language, the result would be a Lisp interpreter. This was a big surprise at the time. Here is what McCarthy said about it later: Steve Russell said, look, why don't I program this *eval*, and I said to him, ho, ho, you're confusing theory with practice, this *eval* is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the *eval* in my paper into [IBM] 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today."[107], page 185.

3.2.3.2 Purpose of execution models

We call an execution model the mechanistic description of a computing execution based on some functional architecture. Such a model supports an explanation of the behavior of a computing system in mechanistic terms. In other words, it explains how computation is carried out by defining a computing system's components, properties, and relationships.

Verifying formal properties is different from investigating why the system behaves the way it does. There are two different tasks. The former task belongs to applied mathematics. It describes abstract computations through formal models by focusing on specific properties. The latter is left to the epistemologist and is the question the philosopher begs when asking what a computer is. It requires something other than task-oriented formalizations of properties abstracted away from any physical mechanism. What the epistemologist needs to make sense of (the overall behavior) belongs to another level of abstraction.

Computations and their models belong to a level of abstraction independent from implementation detail. Computations, as already coined, are "medium-independent" [134]. On the contrary, to have a model of some execution belongs to a lower level of abstraction, where minimal references to the devices that allow the execution are made. Still, there is no need to dig into fine-grained implementation details to make sense of computing behavior in mechanistic terms. The Turing Machine model for classical computation exemplifies the required level of abstraction.

3.2.3.3 The Turing Machine: the execution model for classical computation

We argue that the TM also provides an execution model. Some would object that the Turing Machine does not describe a mechanism, in the sense that Turing's description of his abstract machine provides no hint about how the tape header works, for example. In other words, the TM could be considered non-mechanistic, arguing that it does not provide a complete causal blueprint. Nevertheless, this does not prevent the TM from being a mechanistic abstraction. In a 1950 paper, Turing defines it as a writing mechanism [277] (more comments on that aspect can be found in [141]). The Turing Machine is not a full-blown description mechanism but a mechanistic sketch.

The way the tape head works is abstracted away. But the TM still presents minimal components of a functional architecture: the tape, the tape head, and the state register. Each of the components has some properties. The tape is of infinite length and divided into cells; the tape head can read three types of symbols (0, 1, "blank"), erase them, and replace them with another symbol (0 or 1). The state

register contains n number of states. The relationships between the components and how computation can be executed on such a functional architecture are described following an instruction table. The TM's instruction table is a basis for describing in mechanistic terms the transition from one cell to another. It contains a transition instruction for each state, defining the action to be executed (reading/writing action), a move to the right or the left of the cell, and a new state to be entered. Similar concerns are developed by Bozşahin [39]:

> "The best theory to date for computability, that of Turing (1936), is an abstract mathematical object in the form of an automaton, and even in the abstract form it is physically realizable because it has primitives which are not reduced to other operations, and whose terms are quite simple and clear: move left, move right, change state, read, and write."

As already pointed out, the physical implementation of a Turing Machine is not straightforward. It requires the addition of physical devices, e.g., a clock, to control the tape header (some examples of such realization of TMs can be found, the most famous being achieved in 1972 by Washington University professors Wesley Clark and Bob Arnzen, allegedly known to have built the first physical version of Turing's machine [10]. The fact that implementing the TM abstraction requires additional care about implementation details does not make the TM abstraction non-mechanistic. A complete blueprint is not necessary to build a mechanistic abstraction. Flight mechanics would not tell how to build an airplane from A to Z, and still be an accurate mechanistic description of how a plane comes to fly. The point is that just because a physically implemented TM does not match the TM abstraction, it does not prevent the TM abstraction and its realization from being mechanistically equivalent. Given the mechanistic description of the abstract TM, one can make sense of the execution behavior of a physical TM.

On the contrary, other models of computation, like lambda calculus, do not reference a functional architecture and a mechanistic description of the execution. The singularity of the TM, among other formalisms, has already been pointed out:

> "Showing the step-by-step progress of a function's concrete state in excruciating detail was uncommon. We were used to the global views of Frege and the lambda calculus of Church. It led to an understanding of quite complex tasks, and crucially, at the same time showing transparently that it happens without a concomitant increase or complication in the internal mechanism." [39]

[10] Clark called it the TOWTMTEWP — "The Only Working Turing Machine There Ever Was Probably". See `https://www.thehenryford.org/collections-and-research/digital-collections/artifact/449880/#slide=gs-423201`

3.2.4 Positing the concept of execution model among kinds of computational explanations

Thinking about and proposing relevant levels of description for a computational phenomenon and its implementation is far from being a novelty, especially among accounts of concrete computation [11] in cognitive science [94, 260]. Therefore, in this section, we must position our level of analysis among others that have been classically proposed in the literature to describe computers or computational systems in the broad sense (natural or artificial systems that are assumed to process information like a computer). The question we want to answer (how to describe an interactive computing system) implies a well-known epistemological questioning: given that a computational system is both a physical system describable in physical terms, as well as the implementation of a given abstraction, to which level(s) of description must we refer to produce a satisfactory explanation of the phenomenon? The difficulty is to agree on the criterion of the satisfiability of the explanation.

We believe that at least three criteria are proposed in the literature and define at least three types of approaches for a computational explanation, motivating the choice of the relevant level(s) of description: algorithmic satisfiability (i), functional satisfiability (ii), causal satisfiability (iii). Our execution model aims to satisfy both functional and causal satisfiability.

3.2.4.1 What an execution model does not satisfy: an algorithmic focused explanation

Algorithmic satisfiability is typically represented by Marr's framework of analysis [12] [171] where the explanation is defined on three levels of analysis. In the middle is the level of algorithmic description, the intermediate between the description of the computational level (the computation that one wants to explain) and its physical implementation. In this framework, a good explanation is essentially based on the description of the algorithm, which is given a crucial place to operate the junction between the computational level and the implementation.

An execution model articulates the computational level with an intermediary level between Marr's algorithmic and implementation levels. That intermediary level

[11] Concrete computation refers to computation carried out by physical systems, hence the adjective. For an introduction to concrete computation, see Piccinini's article in the Stanford Encyclopedia: `https://plato.stanford.edu/entries/computation-physicalsystems/`

[12] Marr's work on visual perception, e.g., presented in his book, *Vision: A Computational Approach*, has been influential in analyzing complex information processing systems. See McClamrock's paper [176] for a synthesis of Marr's framework and criticisms.

is twofold: it adds a layer of mechanistic description to a functional architecture. It is less abstract than the algorithmic level in that it tells something about how computation can be executed. But it is more abstract than the implementation level because functional architecture does not commit to the specific description of a piece of hardware. It identifies functions than can be carried out by different hardware pieces depending on processor types or technology maturity.

3.2.4.2 What an execution model satisfies

Functional satisfiability through a functional architecture

Another approach looks for a satisfying functional explanation. It focuses on identifying the functionalities of the component parts of the computational phenomenon analyzed as a mechanism, such as Piccinini's computational account [225]. In that case, the focus switches from an algorithmic analysis to identifying the relevant component parts and the assignment of functions to them. Our execution model satisfies such a functional approach. The components of our execution model are singled out by their function and serve to identify a functional architecture à la Pylyshyn [234]. Regarding the level of analysis, the functional architecture, which is the basis for an execution model, is the description of the blueprint but without reference to concrete pieces of hardware. Functional architecture is distinct from the more concrete notion of "architecture" that can be found in the philosophy of computing [39], in which case "architecture" refers to some real hardware pieces. A transmission system in a car, for example, is a function that we can define as the transmission of energy from the engine to the wheels, but which can be realized by different kinds of hardware. An execution is supported by a functional architecture but not reducible to it. As the label coins it, an execution model is about the description of execution, not merely architecture. With only a functional architecture at hand, one cannot describe the very mechanism by which computation is carried out. It requires an additional description of the interaction between components identified as part of the functional architecture.

Causal satisfiability through a mechanistic description

A third criterion, not incompatible with the functional one, has motivated other types of analysis of computational phenomena and the search for complementary levels of description. This criterion is that of a satisfactory causal explanation. The idea is that a purely formal description does not produce a complete causal explanation. The objective is to define abstract levels of description closer to the implementation level to account for non-calculative phenomena which are also relevant to the production of computational behavior.

This criterion motivates many mechanistic approaches to computational explanation. The idea common to these approaches is distinguishing between formal

computational models and computational mechanistic explanations, arguing that only the latter can provide a causal description of the explanandum. Miłkowski's approach is probably the closest to our stance in this chapter [192]. In several papers, Miłkowski defends his view about the non-mapping between a model of computation and a mechanistically adequate model [188]. The point is that causal organization at the implementation level cannot be told by a formal model of computation. Scheutz [253] has opposed computational and causal complexity in the same line of thought. Similar concerns can be found in Klein's work about process synchronization in neural networks [134].

It is worth fleshing out here the distinction introduced by Miłkowski between models of computation and computational mechanisms [189]. The lesson drawn is that formal models of computing systems do not provide us with the appropriate and complete level of description to build an explanation, which is expected to identify the relevant mechanisms at stake. More precisely, an explanation for computing phenomena requires bridging a high-level description of a computation and its blueprint [187, 190]. The approach is based on the standard of mechanistic explanation in science, coupled with the idea that a computational process is intrinsically mechanistic:

> "Computational explanations, according to the mechanistic account are constitutive mechanistic explanations: they explain how a mechanism's computational capacity is generated by the orchestrated operation of its component parts. To say that a mechanism implements a computation is to claim that the causal organization of the mechanism is such that the input and output information streams are causally linked and that this link, along with the specific structure of information processing, is completely described." [189].

If one is looking for a mechanistic explanation of a computing process, Miłkowski argues that a model of computation may be insufficient. The reason something is missing is that a model of computation is not strongly equivalent to a mechanism:

> "There are two ways in which computational models may correspond to mechanisms: first, they may be weakly equivalent to the explanandum phenomenon, in that they only describe the input and output information, or strongly equivalent when they also correspond to the process that generates the output information." [190]

The difference between strong and weak equivalence captures a difference in causal completeness. The formal models of computation are on the side of models that

are weakly equivalent to a mechanism: "formal models cannot function as complete causal models of computers. For example, to repair an old broken laptop, it is not enough to know that it was (idealizing somewhat) formally equivalent to a universal Turing machine." [190]. Thus, Miłkowski invites us to consider a new project in the philosophy of computing: "it is necessary to acknowledge the causal structure of physical computers that is not accommodated by the models used in computability theory" [187]. To the best of our knowledge, such a project to account for interactive computing has still not been carried out to identify the mechanisms at stake. If philosophers of computing were to proceed in that direction, two criteria for a good explanation of a computer proposed by Miłkowski could offer some guidance. First, such an explanation should be complete, in the sense of a complete causal model where causally relevant parts and operations are specified [189]. Second, a good explanation for computing should explain the system's competence: "By providing the instantiation blueprint of the system, we explain the physical exercise of its capacity, or competence, abstractly specified in the formal model" [189]. For example, it would be necessary to be able to explain in mechanistic terms what the behavior of an oracle corresponds to. This would be equivalent to explaining which mechanisms allow data arrival and how the machine can react to it.

The notion of execution model that we propose is not intended to be a complete causal explanation of the reaction of the computer system and leaves a few black boxes, just as the Turing machine is a form of mechanistic description, but without explaining how the mechanism of activation of the tape head works. As such, we will say that the execution model is a mechanistic schema, as it is defined in the literature [161, 187, 190, 228]:

> "Whenever the causal model of the explanatory focus of the mechanism is complete with respect to the explanandum phenomenon (note: not complete in an absolute sense), the model is a mechanistic how-actual explanation; if the model includes some black boxes whose function is more or less well-defined, it is a mechanism schema; otherwise, it remains a mechanism sketch." [190]

To be more precise, following Machamer, Darden, and Craver [161], the definition of a mechanism schema goes as follows:

> "A mechanism schema is a truncated abstract description of a mechanism that can be filled with descriptions of known component parts and activities. (...) Schemata exhibit varying degrees of abstraction, depending on how much detail is included. (...) Degrees of abstraction should not be confused with degrees of generality or scope."

What we call an execution model abstracts away from some details but is still legitimate as mechanistic abstraction, following Boone and Piccinini [36], since it identifies the components, their properties, and relationships producing the phenomena. The TM abstracts away from how the tape head is controlled. The interactive execution model that we will describe in the next Chapter 4 abstracts away from precise details in memory registers. It still constitutes an overview of how the execution (of classical computation in the case of TMs, of interactive computation in the case of an interactive execution model) is carried out.

3.3 Summary

The chapter claims that new computing practices, as exemplified in interactive programming, cannot be explained within a formal frame and require a mechanistic description. We propose the concept of an execution model as a candidate for such a mechanistic description. We argue in that respect that a twofold model for computing (carrying both a model of computation and a mechanistic description of execution) has ceased to exist. Therefore, there is no support anymore for a mechanistic explanation of today's interactive computing systems. The TM, historically, was such a two-fold model at the time and had provided philosophers with a hint about computing behavior during execution.

In reducing the question of interactive computing to that of super-Turing Machines (as shown in Chapter 2), theoretical computer science does not provide epistemologists with an abstraction to build a general execution model. Instead of a general execution model for interaction, one finds: (i) either, as we have seen in the state of the art of Chapter 2, proposals extending the TM or speculative proposals of paradigm shift; (ii) or, as seen in Chapter 3, a set of formalisms covering various properties required to design non purely computational systems (concurrency, data streaming, synchronization...). Nowadays, formal accounts of computing properties have their own purpose, and their tasks are becoming increasingly specialized, focusing on the formalization and verification of specific mathematical properties. Formalisms are not looking for an explanatory account of computers through an abstraction describing the execution in mechanistic terms.

Focuses on (i) and (ii) deprives us of an actual account of interaction for several reasons, from a theoretical and practical point of view.

From a theoretical point of view, proving the reducibility of a *model* A to a *model* B (when both models represent the same *system* S), may not tell interesting differences between A and B. For example, it may not tell if one of the models captures more fine-grained aspects of S. In other words, there is no guarantee that we understand the specifics of an interactive computing system by referring to the proof

that a formal equivalence is possible between an interactive model of the system and the TM. Another problem is that the various formalisms referring to diverse properties of interactive systems do not constitute a unified and general model of interaction that could amount to the counterpart of the TM for interaction. Moreover, the existing formalisms do not cover with the same interest all the properties of interactive systems. Formalisms aiming at the verification of graphical properties [48, 49, 212, 213], for example, have not been the subject of as much work as the formalization of process concurrency or synchronization.

From a practical point of view, proving reducibility does not allow making explicit the constraints encountered in the exercise of interaction programming (Chapter 1).

Defining an adequate mechanistic description and an execution model for interaction is still challenging. Our proposal will be introduced in the next chapter.

Chapter 4

Conceptual proposal: an execution model for interaction

In Chapter 3, we have characterized the opposition between *formal model* and *explanation*, borrowing from the analytical philosophy of science. The previous chapter has motivated the idea that when looking for an explanatory abstraction to characterize interaction, it is promising to look for an execution model. Such an execution model cannot be merely formal but needs some mechanistic account. We showed that the TM was such an abstraction for computation and that it is left open to define the counterpart of the TM for interaction.

Let us add two caveats to this point. First, the explanatory abstraction we look for cannot embrace all the abstraction layers involved in a computing system, ranging from voltage firing and logical gates to high-level languages and libraries. As already developed by Lee [1] [147], there is no way to make a computing system understandable to a human from layer A to Z within one explanation only, because of an overwhelming number of details. Therefore, we look for a trade-off to build an understandable and explanatory abstraction. Second, what we propose is not a full-blown execution model. We present its *requirements* and derive from them three minimal components. We discuss the link between our execution model and existing interaction languages and framework. In other words, we think our execution model can serve two purposes: a general explanatory purpose and a reflection on interaction-oriented programming languages.

Here is how the chapter is constructed more precisely. We successively present:

- The requirements for an **execution model** to support the *understanding of interactive systems*. From the identified requirements, we propose compo-

[1] See in particular chapters 3 and 4 in *Plato and The Nerd* [147]

nents for such an execution model and an associated mechanistic description — Section 4.1.

- An investigation among existing programming languages to test whether our proposal supports a **typology** among interaction frameworks and languages — Section 4.2. In other words, we will see how our proposal can account for and classify the frameworks and languages dedicated to interaction.

4.1 The execution model for interaction

We are looking for the counterpart of the TM for current interactive computing: explaining the very possibility of execution. Therefore, complete independence from the medium cannot be preserved. We need at least abstract references to what supports execution. As explained previously in Section 3, an execution model is an intermediary representation that bridges the upper layer and the physical implementation at the processor level, providing a mechanistic account of how interactive computing behavior is carried out. A minimal execution model would at least conceptualize the necessary components that allow execution. In the case of current computers and their interactive behavior, there are specific interesting phenomena to be explained that are not accounted for within the TM execution model. Some extra phenomena belong specifically to interactive devices, where human agents interact with the system. From the interviews (See Section1.1 in Chapter 1), the software models (See Section 1.2 in Chapter 1), and the literature review on interaction (See Chapter 2), we propose to conceptualize what these phenomena are and argue they are minimal conceptual requirements for interactive systems.

4.1.1 Specifics of an interactive execution model

Specifying computational algorithms is only one task among many when programming interactive behaviors, as we saw in Chapter 1. What is relevant is also defining what will trigger a computational mechanism or a state change for the machine.

For example, one may wish to program the following interactive behavior as shown in Figure 1: (i) as long as the mouse button is inactive, nothing happens; (ii) movement of the mouse triggers movement of the cursor; (iii) pressing and holding the mouse fills the memory at position $y \times width_screen + x$ (which then translates into the display). The definition of this action sequence has a global order (i, ii, iii). However, it is not an actual procedure or a calculation in the classical sense. A computation is indeed executed whenever it is required ($y \times width_screen + x$), but

it is not sufficient to account for the expected behavior. From an interaction point of view, what is important is that the coupling between mouse activation and position display is specified and that this constitutes a reliable and deterministic behavior. The expressiveness of an interactive language must therefore be measured by its ability to express such relationships. We propose to detail three components that seem essential to the expression of interactive behaviors: *causality*, *transduction*, and *physical time*. We suggest that each phenomenon should find its mechanistic counterpart within the execution model.

4.1.2 Minimal requirements

More specifically, we argue that a general mechanistic explanation of interaction should account for the following:

- how the internal computing processes can be triggered by external events; in other words, how **causal relationships** between processes as specified by the programmers can hold;
- what allows physical phenomena to interact with computing processes; in other words, what role **transduction** plays;
- how the internal computing processes can be organized in time and respond to timer activations, which supposes a reference to **physical time**.

4.1.2.1 Expressing causal relationships

First, as the interviews and the literature survey suggested, interaction programming is about the programming of causal relationships. Specifying an interactive behavior of a programmable machine relies on the description of relations between the occurrence of events and the triggering of various processes, possibly computational, within the machine. This has been theorized as an "essential feature of interaction" [235]. Although the concept of "causality" is not always used explicitly in the literature, we showed in Chapter 1 that it helps conceptualize significant issues described by programmers. The causal requirement for interaction is notably described by Myers in the following term [207]:

> "Any large, complex application contains thousands of interdependent relationships. For example, a graphical application must deal with the relationships arising from laying out objects, displaying feedback for input operations, and keeping the view consistent with the underlying

> data they represent. (...) Constraints provide a convenient way to specify relationships and have them automatically maintained at runtime by a constraint solver."

The causal vocabulary is pervasive in the literature when describing an interactive system: [126, 136, 154, 170, 205]. The possibility of such causal relationships between processes is the mechanism to explain. The need for such an explanation has been pointed out in the literature, for example, in the "Anatomy of Interaction": "We believe that the major fault of current approaches to programming interactions is that they do not account for **how interactions come to be**" [19] (See Section 1.2.3 in Chapter 1), or in the Reactive TM community: "In order to mimic site machines, a Turing machine must have **a mechanism that will enable it to model the change of hardware or software by an operating agent**" [151] (see Chapter 2).

The programmer often needs to specify when to stop or restart processes. This is why an explanatory abstraction should have basic mechanisms that allow controlling the life of a process from triggering events. The concept of "event" is understood here in the general sense of something that happens. This can be an event internal to the computer system (such as the end of a computation process) or external (such as a mouse click).

This dimension is absent from the Turing machine: there is no way to describe the launching of the execution in reaction to events. It makes no sense to ask when the tape header starts reading and rewriting a box. For the Turing machine, all inputs are given before the execution starts. It is the succession of computation steps and the final result on the tape that matter. Similarly, the classical model does not allow the expression of the pausing or restarting of the automaton: it does not make sense for a computational algorithm. One could object that the Turing machine with oracle models the pausing and the continuation of the calculations. There is, however, a major difficulty with this extension. Indeed, when we talk about interaction, we do not describe an automaton that, at a stage of its execution, would ask an oracle (or any other abstract representation of an external agent) for a new symbol. We want to specify how an external process can interrupt or launch a machine process. In other words, interaction presupposes an inversion of control: it is no longer a Turing machine that controls the course of its execution but rather the external environment that controls the execution flow. This requirement has been discussed in the literature. In Chapter 1, we saw that Suchman [256, 271] insists on the possibility of interruptions to reflect on that matter. In the Reactive Turing Machine community [151] (See in Chapter 2, Section2.2), that aspect is also put at the forefront, but without an account of the mechanism allowing the writing on the input and output ports:

> "At each step, the machine reads the symbols appearing at its input ports. At the same time, it writes some symbols to its output ports. Based on the current context, i.e., on the symbols read on the input ports and in the 'window' on its tapes, and on the current state, the machine prints new symbols under its heads, moves its windows by one cell to the left or to the right or leaves them as they are, and enters a new state." [151]

Remember that in the formal account proposed by the Reactive Turing Community, the oracle is the abstract concept supposed to ensure the arrival of new data. But an oracle is not a mechanism and does not provide an explanation on the very possibility for a machine to launch, pause and resume the reception of new data[2].

High-level mechanisms, such as event loops, wait continuously for new inputs. In current computer architectures, some mechanisms change the state of the machine upon the arrival of new data during execution. In many processors, they are interrupt mechanisms, launching, pausing, and resuming processes upon arrival of new data — but they can take other forms. When an interrupt is requested, the running process is suspended, some information is saved, and a pre-defined code called a routine is executed. For example, moving the mouse or pressing a key on the keyboard causes an interrupt, which in turn calls a routine. These interrupt handlers allow the reading of the mouse position or the value of the pressed key. What has been read is then copied into memory. At a low level in memory, the connection between the external input arrivals and the computing system may correspond to changes in specific memory registers. Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer. A current alternative to interrupts is *polling*. With polling, the CPU steadily checks whether an I/O device requires something to be processed. Whether it is polling or interrupts, the fact is that in their absence, any form of interaction with arriving data is impossible. Without such a mechanism, it cannot be told how the change of symbols on the tape's cell can involve any change in the system. Thus, at the level of the abstract model, this type of mechanism appears as a necessity, either by steadily checking the value in memory (in the case of polling) or waiting and reacting in case of a value change (in the case of interrupts).

[2] An anonymous reviewer of a paper we submitted at *Minds and Machines* phrases the idea in more clear-cut terms, telling that an oracle machine has nothing to do with interaction. An oracle machine is an infinite table lookup that solves some problems, including undecidable problems. For example, an oracle machine's table can determine whether a particular TM halts for a particular input, thereby 'solving' the halting problem. However, this has nothing to do with interaction.

4.1.2.2 Referring to transduced data

Second, interrupt management mechanisms *or any mechanism that would ensure causality management* between external and internal processes are insufficient to ensure the link between the physical world and the computational processes of the machine. An additional mechanism is required to convert physical quantities into digital data: transduction. The importance of transduction for interaction computing has already been commented on in theoretical computer science [152], the epistemology of computing [162], and HCI [3, 19, 50, 56, 213]: "For computers and software to become mediators of human action, they need to be able to respond to the outside world. This is broadly the purpose of interactions as implemented in code: to transduce changes at some locations to changes at other locations" [19].

It has also been argued that a low-level model for transduction is crucial for designing fine-grained interactive systems [3, 56]. Some inputs can be described by a simple Boolean value, indicating the presence or absence of a signal (such as a mouse click). Others can have a more complex structure, such as the continuous evolution of a physical magnitude (light or temperature, for example). A fine-grained interactive system enables the programmer to easily connect and disconnect transduction sources [19].

The Turing machine, modeling the inputs as a sequence of symbols, does not allow one to account for the variety of physical phenomena that can cause computational processes. The objective of a Turing machine was indeed to give the result of a calculation. Therefore, the size and shape of the inputs were of no importance. However, when the objective is to specify causal relationships between physical and computational processes, it is important to be able to express the structural and temporal properties of the physical processes. This means that a transducer is not only the digitization of an analog signal, it must also preserve and transmit the *causal* structure of the physical phenomenon [3]. We talk about "causal" structure to refer to many transduced physical phenomena, where the programmer needs to consider the periodicity of phenomena. For example, refreshing a frame or adjusting video frame rates requires that the programmer calibrate the reception of data. The issue was commented on at length by I5, I8, and I11.

In HCI, the challenge of transmitting information is increased with the growing number of input devices, and multimodal interaction [212] between users and computer, as commented by several interviewees (I5, I7, I11): e.g., hands-discrete inputs" [125], "hands-continuous inputs" [125], other body movements (like head position or direction of gaze), voice, virtual reality inputs.

4.1.2.3 Expressing measurement of physical time

Third, in the classical theory of automata, physical time is ignored [3]. There is only a notion of logical time, reduced to an order of a sequence. Thus, the Turing machine, or any derived abstract machine, allows only specifying a sequence from an input to an output step by step. It is not the physical time but the number of computational steps that is the reference. The physical time it takes to execute a step (i.e., a step in the computation) does not exist for the model [158]. In the field of distributed systems, this lack of a physical notion of time has proven to be problematic. The solution has been to reduce the notion of time to a notion of order by using logical clocks or timestamps [140]. However, programming interactive behaviors requires a richer notion of time: a notion of physical time, i.e., duration, measured by a physical unit. A gesture-based (post-WIMP) drawing tool or any interaction technique cannot be implemented without the description of timing aspects to represent the quantitative temporal evolution of the interaction technique [50, 213]. Interviewees commented on the issue of adjusting timers, e.g., to program a fine-grained animation adapted to human perception (I2, I5) or to await the arrival of data (I1, I5, I8, I11).

Furthermore, the systems we are talking about interact with humans and involve human perception, which is sensitive to duration. Thus, the display of a message, for example, cannot be too brief and must appear long enough to be readable by a human agent. Hence the programmer needs to be able to express a duration, here to specify the time after which the process can be interrupted.

A specific case of interactive systems like critical embedded systems presents another kind of sensitivity to duration. As commented by Lee [144], the computational processes involved are themselves the result of physical processes internal to the machine, and these processes take time. Usually, the time it takes for a processor to execute a computational process can be abstracted away. However, once the hardware gets "old" and too slow for more demanding tasks, this execution time becomes critical.

In any case, the problem with duration is that it cannot be reduced to some steps or stages in a computation: we do not know how long a stage takes, and we cannot assume that each computation stage can be executed in an equal time. The only way to specify the duration is to refer to a physical process that can quantify a duration: this is the case with the frequency of an oscillator. At the level of an execution model dedicated to interaction, this implies including a reference to a physical clock, which allows specifying a duration.

[3] As for timed automata, we point out in Section 3.2.2.3 related issues for a mechanistic explanation.

It is true, however, that there are interactive devices that do not require clocks (there even exist CPUs with a clockless architecture). However, we argue that they are a subset of interactive devices rather than the general case. Most interactions with humans require some notion of time elapse. References to physical time measurement would be required to define rich interactions (animations, long press, double click, production of sounds, etc.).

4.1.3 Components

To build a mechanistic interactive execution model, we need to identify the warranted components of the mechanism, their relationships, and their properties. In the following, we define three minimal components derived from the previous requirements to account for the previously mentioned relevant phenomena. We introduce each component and describe its properties and relationships with other components.

1. **A source component**. It is enough to say that an execution model must have a set of causal sources at a higher level. A process becomes a source when it triggers a causal relationship, stored and executed by the causal orchestrator. Sources can have four origins: (i) generated by internal software processes (e.g., a computation, or the assignment of a result in a memory cell). (ii) generated by transduction phenomena (Analog-to-Digital conversions), and (iii) generated by the internal elapse of timers. As opposed to hardware or physical processes, software processes can be programmed/changed by a programmer on a general-purpose computer. They may implement computation in the sense of the TM.

   Source components are explicit in some views of interaction, such as the one supported by Whizz [56], the ICON toolkit and its so-called "input configurator" [84], also integrated in the MaggLite toolkit [122].

   Any source can change the state of the machine. The state is a steady but modifiable model of the world that records past transductions, elapses, or results of internal software processes [66]. It can be implemented in the form of memory cells, which become sources when modified. We could have considered states as a first citizen component. However, we decided to focus on the *specifics* of interaction. States are also a core feature of the Turing machine. Therefore, since that feature is common to interaction and computation-oriented systems, we chose not to detail it.

   Properties of sources. Causal sources have a structure in time and space that they can transmit to the causal orchestrator. A formal description of

a causal source should provide a model for the organization of its parts and characterization of its associated magnitudes. A mouse, for example, can be described as the composition of a 2D signal continuously sending displacement quantities, with a set of occasional valued signals for the buttons.

A fine-grain theory of interactive devices should provide an ontology of the varieties of causal sources [3, 56, 125].

Relationships with other components. Causal sources are connected directly to the causal orchestrator (the second component, presented below), which ensures the right reactions are triggered by the input arrival. Signals sent by the expiration of timers are among the possible causal sources, which makes it possible to express duration in interactive computing systems.

2. **A causal orchestrator component**. When programming interactive behaviors, a requirement is that the entire system responds in a deterministic way to the unpredictable arrival of events. That is, the interactive machine must ensure that the order of execution complies with the causal relationships specified by the program. This component is reminiscent of what is labeled as "constraint solver" in Garnet [207], and the term "orchestrator" has already been used in a similar sense in the literature [30, 235]. This has been a concern in the development of new tools for interaction, for example, MaggLite [122]. We argue that the role of such a component is hardly modeled by the behavior of the tape head in the TM. In a classical Von Neumann architecture, it is the role of the program counter to ensure the right ordering of a specified sequence of instructions. For an interactive machine, the abstract component that warrants the right causal ordering should be conceptualized as such, and we suggest labeling it as the "causal orchestrator". In current processors, such a mechanism exists in the guise of an interrupt vector table. The operating system (OS) also partakes in this causal orchestrator role. In that respect, the Portable Operating System Interface (POSIX) exemplifies a way to make the managing of the causal orchestrator available to the programmer. POSIX is a family of standards defining both the system- and user-level application programming interfaces (API) and command line shells and utility interfaces for software compatibility (portability) with variants of Unix and other operating systems. The core services provided by POSIX could serve as detailed examples of the kind of instructions required for a causal orchestrator: e.g., process creation and control, signals (that send messages to a running program to trigger specific behavior, such as quitting or error handling), I/O Port Interface and Control, semaphores, thread creation, control, cleanup, scheduling, synchronization.

Properties of the causal orchestrator. The causal orchestrator can start, stop, or resume processes, following causal instructions and establishing the

right connections between causal sources and connected processes. It should also provide flexibility to allow the composition of new connections, e.g., between new input devices and new interaction techniques.

Relationships with other components. The causal orchestrator receives the structure of the causal sources. Among causal sources are time values transmitted by the clock component. The causal orchestrator ensures the sources trigger the wanted processes, which may be computational (e.g., recalculation of a value) or physical (e.g., launching a new timer).

3. **A clock component**. We argue that clocks are not like any kind of source but deserve a specific status. The reason, we think, is that clocks cannot be replaced by another kind of source in order to complete their tasks. Clocks are involved in designing an interactive computing system when one needs to specify timing instructions. By "time", we refer to the physical time or an elapsed time, not to order. Therefore, a specific representation of time is required in the model. Clocks are also involved in the orchestration of polling. It looks like internal clocks can hardly be replaced by another external source that could have the same roles.

   Properties of the clock. The clock component should provide both a timer facility, i.e., a component that fires an event when a scheduled duration is expired, and a clock facility, i.e., a component that sends a periodic signal. It is difficult to determine which component is the most fundamental. On the one hand, one can build a clock by establishing a causal relationship between the end of a timer and its restarting. On the other hand, one can build a timer from a clock by establishing a causal relationship between a specified number of clock ticks and the stopping of the timer. However, the most basic component enabling time measurement at the hardware level is an oscillator.

   Relationships with other components. The clock component feeds the source component (a timer's expiration becomes a source).

4.1.4 Refining the causal orchestrator with dynamicity concerns

A typology refinement could be the dynamicity of causal relationships. Three levels can be identified beyond a **level 0**. Level 0 would amount to having no possible causal relationships programmable between processes.

- **Level 1. Causal relationships are static**. Causality is expressed in the following general fashion: "any update of the value of x, updates the value of any entity connected to x". Static here means that the network of connections

between entities does not change during execution; thus, the causal path is immutable. A further distinction could be made here depending on the possibility of changing the behavior of each entity according to its history (stateful *vs.* stateless). This level of dynamicity corresponds to what is defined in Garnet [207] as a "one way constraint" or binding in Smala.

- **Level 2. Causal relationships can be modified at runtime**. This type of dynamicity basically refers to the possibility of switching from one causal path to another one during execution (some are activated, others are deactivated). In the context of a set of causally connected processes, this means that the execution of one process will change the causal connection between some processes. Some causal links are (de-)activated during execution. For this class of dynamicity, the processes and the topology of their causal links is defined at the beginning of the execution, and the dynamicity comes with the possibility to (de-)activate causal paths during execution.

- **Level 3. Causal relationships can be created or deleted during execution.**

  The third and most powerful type of dynamicity allows a dynamic change in the causal structure, that is, adding or deleting components and causal paths during the execution of the program. Such an evolutive causal structure appears necessary in many applications where one wants to react to the appearance/disappearance of external processes. A sublevel could be introduced within level 2. Let us call it **level 2+**. This would allow taking into account a type of dynamicity required when a programmer wants a defined causal relationship to apply to newly created objects. We could mention here as an example a case mentioned by two of our interviewees (I1, I5), coding two interfaces for Air Traffic Control, one displaying in real time planes on an airport, the other the strips with information corresponding to the monitored flights. They needed to apply the same code to each new incoming flight. In that case, it is not the causal relationship per se that changes, but only the *relata* that are modified. In other words, it is not the meaning of the arrow in $A \rightarrow B$ that changes (activation, deactivation, adding, deleting) but A and B that change, replaced by a dynamic reference to newly created processes C and D.

4.1.5 Mechanistic description

In Figure 4.1, we propose a representation of our execution model. We add extra comments to detail the links between each element.

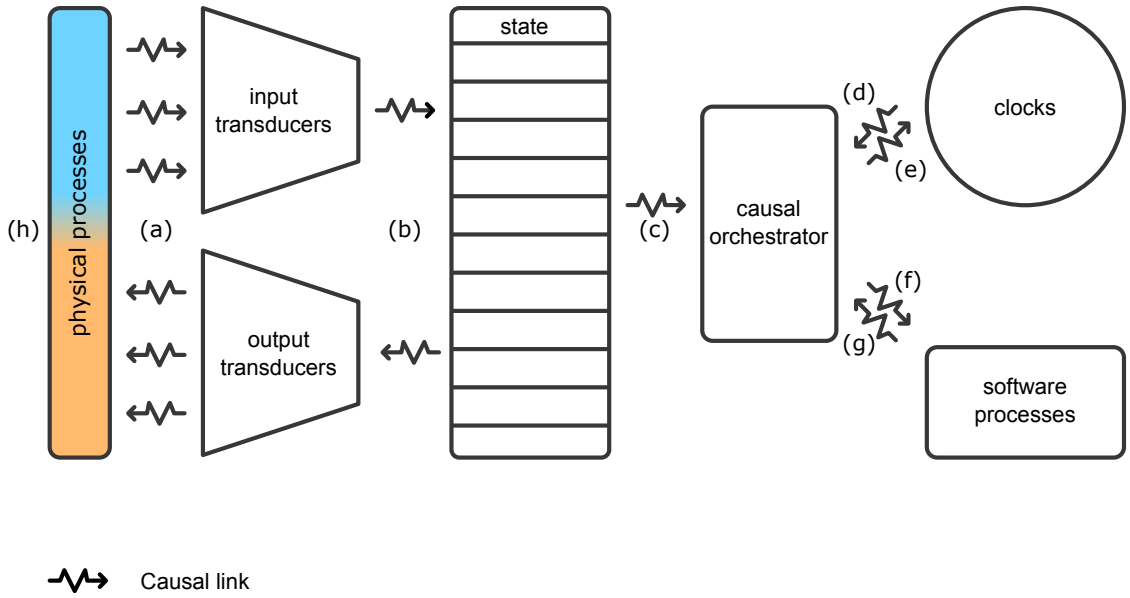

Figure 4.1: Representing a general interactive execution model

First of all, to avoid confusion, the arrows in the figure specify causal links, not dataflows. As such, they specify meta-causal relationships: they are not the causal relationships managed by the causal orchestrator but rather the causal relationships that implement the mechanisms of the machine.

(a) Transducers are physical causal links that ensure the translation of physical processes into digital inputs. **(b)** The arrival of new transduced data changes the state of the machine. **(c)** As commented before in section 4.1.2.1, the machine may react to the reception of new data and trigger the causal orchestrator. The causal orchestrator then ensures that the causal relationships specified by the programmer hold. The causal orchestrator keeps count of the causal relationships specified by the programmer and guarantees their ordered, unique execution **(f)**. In turn, the software processes may trigger other causal links through the causal orchestrator **(g)**. Since some relationships depend on timing constraints, the causal orchestrator activates clocks and timers **(d)** and responds to their ticking **(e)**. As an interactive execution is a feedback loop involving a user, transformational processes are, in turn, transformed **(b)** into physical outputs available to the user through transduction **(a)**. Inputs **(h, orange)** may trigger changes of outputs **(h, blue)**, and outputs possibly become inputs during the ongoing execution **(h, orange-blue)**, implementing the inter-action between the user and the machine.

A few additional comments can be made:

- **(c)**: In some concrete implementations, the machine mechanisms may rely

here on interrupts.

- **(c)(e)**: Alternatively to interrupts, another mechanism like polling may ensure the causal link between the reception of data and the causal orchestrator: a clock steadily makes the causal orchestrator poll the state and trigger an execution cycle.

- **(h)**: Outputs and inputs are blended within a single block along a colored spectrum, to represent the fact that inputs are the source of outputs but that outputs in turn can affect future inputs. Examples include the apparition or deletion of invisible picking zones implementing a transient spatial mode or movement constraints for a phantom haptic device.

Now that we have a sketch of a mechanism supporting interactive execution, we can test whether it allows a mechanistic description of the drawing app's behavior, introduced in our touch-based drawing application example (Figure 1).

We could apply the following mechanistic description. There are several sources. These are first the user inputs (more specifically, their dimensions/properties — the pressure of the moving gesture, coordinates of the tap on the screen) and expired timers. A causal orchestrator construes the causal relationships between user inputs, timer expiration, and connected events. The connected processes are here the *draw* and *erase* processes. The mechanistic description goes as follows: the user touches the screen. This triggers the activation of a first source value: a complex source value containing transduced values about the pressure on the screen and the coordinates of the press. A timer is launched (100msec, e.g.). Another causal source is triggered if the timer expires: an event referring to the expiration. If the user, in the meanwhile, has not tapped again on the screen, the causal orchestrator connects the pressure of the touch on the screen and coordinates with the drawing process. The release of the pressure triggers the dynamic creation of an object from the drawing that becomes an object to be interacted with. If the user taps another time on the screen before the expiration of the timer, the causal orchestrator triggers an *erase* event, and the drawing disappears.

Notice that although it could be represented by a finite state machine (FSM), such a description is not reducible to an FSM. The same objection we did in Chapter 3 about timed automata would apply: an FSM could not provide us with an explanation about how the execution is made possible and how the transitions between states work. A lower abstraction, such as the execution model, is warranted for such an epistemic task.

The example also demonstrates why we think the components are minimal. Without one of the three components, any mechanistic description of the phenomena fails. If one takes out the clock component, the difference between a single or

double tap cannot be explained. If one takes out the source component, there is no way to account for the possibility of a timer expiration causing the resuming or stopping of the drawing process. And without the causality orchestrator, the connection between the timer expiration as a source and the resuming or stopping of the drawing cannot be expressed. In that respect, the three components look like the minimal building blocks for an execution model. Each component can indeed be refined into sub-components (e.g., types of causal sources, sub-routines in the causality orchestrator), but for clarity, we prefer not to delineate the model further.

Despite its simplicity and apparent triviality, the drawing example (in Figure 1) we consider is far from being an epiphenomenon among interactive systems. We think such an example captures the core features of interaction: it exemplifies a coupling between a user and computational processes, where the execution is driven by the user's actions. It is an example of the ongoing transformation of outputs into available inputs thanks to dynamicity and graphics selection (picking). Therefore, we think our basic drawing example has the same status as other examples that have been similarly chosen to illustrate essential features of a programming style or paradigm. The "hello world!" program is a minimal example to illustrate program execution; the computation of factorials or the Fibonacci suite usually introduces functional programming, being minimal examples of recursive function calls.

4.2 Linking the execution model with existing interaction languages

We now have an execution model that carries a mechanistic explanation for interaction. Ideally, to be provided with the counterpart of the TM for interaction, we would need a twofold abstraction: an abstraction that gives the intuition about a mechanism but also relates to a set of languages.

We presented a landscape of interaction languages and frameworks in Chapter 1. We can now examine the link between the execution model and interaction languages along two main axes. The first is that of translation or semantics, i.e., how does a well-formed expression in a language translate into the operation of an execution model and vice versa? The second possible axis of analysis is measuring the expressiveness of a given language, i.e., does a language *Gamma* allow the expression of all possible interactive behaviors? In other words, the question is whether our proposal allows a relevant typology of languages/frameworks and accounts for significant differences.

4.2.1 From the execution model to interaction semantics

The first axis is that of translation or semantics, i.e., how does a well-formed expression in a language translate into the operation of an execution model and vice versa?

For example, a causal relationship can be described by the activation of a process following the activation of a source, i.e., $S_{source} \rightarrow P_{rocess}$. In the case of an interaction-oriented language, the translation will be available; in Esterel, for example, we will write something like:

```
input S;
output P;
loop
        await S;
        emit P
end loop
```

With Java Swing, the registering of a listener callback is a causal construct expressed as follows:

```
source.addListener(listener)
```

With Qt, the connection between a signal and a slot is expressed with:

```
object1.signal.connect(object2.slot).
```

In Smala, the binding between the activation of two processes is expressed with `p1->p2`, an assignment with `p1=:p2`, a dataflow with `p1=>p2` and a transition with `s1->s2(event)`.

With SwingStates, the transition between two states is expressed with

```
Transition t = new PressOnShape(BUTTON1, "»menuOn")
```

while with QML a transition is expressed with

```
DSM.SignalTransition {targetState:finalState signal:button.clicked}
```

A synthetic overview presents the other translations in Table 4.1. It is for now a work in progress that must be continued, with corrections and further study. It tries to identify for different languages, from the less to the more interaction-dedicated (degree 0 to degree 3 defined in Chapter 1), what are the expressions of the sources, clocks, and the causal orchestrator. For the causal orchestrator, we try to identify what programming construct encodes causal relationships and what mechanism guarantees the causal relationships hold at a lower level (e.g., a framework or an execution engine).

4.2.2 Interaction expressiveness of existing languages

The second possible axis of analysis is measuring the expressiveness of a given language, given the execution model. It helps conceptualize the concept of interaction completeness: a maximally interactive computing system should allow the expression of all possible behaviors. Some works have been carried out in that direction [19, 44, 49, 126, 132, 133, 212, 213], although they did not choose the description of an execution model.

Given our execution model, we can go back to the landscape and see whether a mapping is possible. First, expressiveness for interaction could be measured with respect to the existence of means of referring to the components from the execution model. Second, to refine the measurement of expressiveness, it would be relevant to evaluate the level of dynamicity of the causal relationships we identified.

The most expressive semantics would allow fine-grained reference to time, transparent access to causal sources and their dimensions, and dynamicity of type 2.

The least expressive languages, such as "purely" functional languages, i.e., without side-effects and reactive extension (pure non-interaction-oriented languages), do not allow the expression of causal relationships. For the languages that allow different levels of dynamicity, we flesh out the following typology and a possible mapping.

Static causal path (level 1): Level 1 refers to languages not allowing causal relationships to change during execution. This class gathers many of the reactive functional languages used for dataflow programming. Data-flow programming [245] emphasizes the propagation of data and models programs as a series of connected entities that work as small computational units. Such causality expression supports dataflows, as implemented, e.g., within the functional paradigm, as in many functional reactive languages [88, 166, 177].

Dynamic causal path (level 2): Level 2 refers to languages allowing causal relationships to be dynamically activated or de-activated during execution. A usual way to achieve level 1 is through the use of finite state machines, as in StateCharts [115], SwingStates [10], FlowStates [11] or in the Hierarchical State Machine Toolkit (HsmTk), [34]. If a state denotes a set of causally connected processes, then a state transition from A to B will deactivate all causal links denoted by A and will activate the ones denoted by B. This is a familiar representation of causality that dates back to nets and event structures [215]. An analog result can be achieved using specific operators in control flow languages, such as Esterel [26, 28]. In that case, condition and action statements are declared: "when p, do emit q, end". State transitions express causality in the following general fashion: "whenever getting into $State_a$, a pre-defined transition to $State_b$ occurs". A dynamic causal path is also allowed by specific dataflow languages, such as Lustre, with the implicit use of state machines [53].

Evolutive causal path (level 3): Level 2 refers to languages allowing causal relationships to be dynamically added or deleted during execution. This is the case in JavaFx, PyQt, or Smala, for example. Such languages raise theoretical issues to be formalized, and some have been addressed [12]. One of the most difficult challenges is keeping a

coherent causal path along execution, that is, removing possible broken links and adding new ones at the right place. In a language like Smala [164], Level 2+ is allowed by the use of a $RefProperty$.

Interaction dedicatedness	Language/framework examples	Causal sources	Clocks	Causal orchestrator	
				Programming constructs	Implementation
Degree 0	Genuine C	instruction, function	oscillator	function call(), ";"	program counter, *jmp*
	Genuine Lisp	function		function call(), *eval*	
Degree 1	C with I/O libraries	function	*wait, sleep*	*select, ioctl, getc, putc*	program counter
	LISP with REPL	REPL		*loop*, <CR>, *eval*, *read*	
	JavaFx	property, event		binding, event listener	framework
	Qt	*signal*	timer, clock	*signal & slot*	
Degree 2	Elm	*msg* (message)	*timer, clock*	*update*	
	Reactive ML	data stream		*loop, await, emit, run*	compilation/static
Degree 3	Esterel, Ceu	data stream	oscillator	*loop, await, emit, when, end*	compilation/static
	Lustre	*node*	oscillator	equation	compilation/static
	SwingStates	*events*	timer, clock	finite state machine, state, transition	execution engine
	FlowStates	event, input slot	timer, clock	finite state machine, state, transition, dataflow	execution engine
	Garnet	*interactor, slot*	timer, clock	*one-way constraint "="*	*constraint solver*
	MaggLite	input slot	timer, clock	*Interaction Graph, Interaction Access Point*, dataflow	
	Tcl/Tk	*active variable*, graphics/widget	timer, clock	*binding*	execution engine
	Smala	*processes*	*timers, clock*	*bindings*, finite state machine, state, transition	Djnn execution engine
	VHDL	*process*, signal	clock	*process, component, case statement*	hardware

Table 4.1: From the interactive execution model to existing languages and frameworks: a typology attempt.

4.3 Summary

We have identified a minimal functional architecture for interactive computers (sources, a causal orchestrator, and clock components). It supports mechanistic descriptions of execution behavior that help make sense of interactive computing phenomena, such as the interaction between a user and a drawing application. This is an intermediate abstraction with a reference for each component to the actual physical device supporting it. It is on board with the concept of functional architecture à la Pylyshyn presented in Chapter 3 but adds two dimensions to it. First, it is extended to describe the architecture of an interactive computer. Second, it deepens the underlying mechanism supporting the function of each component, concerned with execution — not merely architecture. An execution model can provide a mechanistic sketch of the execution. What we call an interactive execution model supports reflection on interaction programming languages and their expressiveness. We have proposed, based on our three components, a typology of existing languages and frameworks dedicated to interactive programming, which suggests at least that our execution model could help reflect on practice.

This chapter conceptualized the role of a causal orchestrator in executing interaction programs. It will motivate, combined with the results of the interviews, the tool presented in the next chapter.

Chapter 5

Practical proposal: *Causette*, interaction techniques to support causality understanding

In Chapters 1 and 4, we gained more insights on the specifics of interaction programming, especially regarding causal orchestration. In this chapter, we want to address the *understanding concerns*, related to causal relationships (see Subsection 1.4.3). We believe that in addition to relying on specific constructs for interaction-oriented programming, developers should be able to follow and understand the *causal chains* [1] to assess whether the code of an interaction will behave correctly in future executions.

We gave in the previous chapter examples of causal constructs. The challenge we tackled here is the programmer's ability to understand such causal chains, notably in FSMs and dataflows. Since interaction depends on external events that might occur in any order, understanding an FSM requires considering multiple causal chains. Similarly, developers must follow the flow of data across multiple code locations to understand a data-flow of connected components. Few tools have addressed these difficulties in past work.

We present the design of Causette[2], a set of four novel interaction techniques for a code editor to help programmers understand the causal relationships. The interaction techniques consist in rearranging causal constructs on demand and bringing together the causal relationships that are far from each other in the source code. In so doing, Causette makes the code representation and its visual ordering of lines consistent with the causal chain being analyzed by the programmer.

We start with a literature survey (Section 5.1) to motivate design choices. This involves looking at how causality has been addressed in interactive programming, how textual

[1] We defined what a *causal chain* over multiple code locationsis in the introduction.

[2] Causette is a pun from the French given name Cosette spelled like Causality, and evoking "*-et" HCI terms such as "widget" or "applet"

code in IDEs is augmented and surveying views of code representation and animations. We then sum-up in Section 5.1 ideations from the interviewees about possible solutions to support interaction code understanding. Section 5.3 presents requirements and design principles. The interaction techniques are presented in detail in Section 5.4 and we demonstrate them in relevant scenarios. Finally, Section 5.6 describes a semi-controlled quantitative and qualitative experiment with 10 professional programmers. The experiment shows that Causette may be more usable than a regular editor for some code understanding tasks. This work suggests that rearranging interaction code may help developers better understand and fix it.

5.1 Literature survey to support the design of Causette

5.1.1 How causality has been addressed in interactive programming

In the field of interaction-oriented programming, the causal challenges related to interaction code have been pointed out [57, 153, 205, 206], but few tools have addressed it. The WhyLine debugger [136, 137] provides developers with answers to a *why?*-question regarding a phenomenon they perceive on a user interface i.e., a causality question. WhyLine provides an answer to a causality question, but during execution only. Many papers have studied developers' needs outside the scope of interaction programming, particularly when debugging [142, 174, 263] and proposed new debugging methods, notably in the field of functional reactive programming [247, 249]. However, few of them have focused on debugging interaction-oriented programs or causal relationships. Aside from the field of programming, some work in information visualization has studied the representation of causality. This includes exploring graphs and diagrams to visualize causality [89, 90, 282], applied to statistics or the modeling of distributed systems. However, typical graph layouts may not be appropriate to follow the control flow of an application, as they force the readers to visually hop from node to node in arbitrary directions. Those limits have been addressed by tools like DA4Java [230] for Java source code. To make the graph more readable, a set of features allows incrementally composing graphs and removing irrelevant nodes and edges from graphs.

5.1.2 Augmenting textual code in IDEs

Several issues with program understanding motivate work on textual code augmentation. One issue is related to file-based IDEs. The literature highlights how IDEs lack effective support to browse complex relationships between source code elements. Developers are often forced to exploit multiple user interface components at the same time [138], making the IDE "chaotic" [186]. To prevent time-consuming navigation between files,

CodeBubbles [40, 41, 240, 241] offers an integrated development environment for Java. With CodeBubbles, programmers can build working sets composed of code fragments (like methods, small classes, notes, documentation, etc.), displayed in a separate bubble or lightweight window. Programmers can rearrange the layout of the bubbles to create a logical context, e.g., to support navigating from a caller to a callee. VSCode offers a functionality called CodeLens [178]: a piece of actionable contextual information (from different files) interspersed in the edited code. It differs from Causette in terms of interaction technique (pop-up window), type of information (e.g., editing history), and targeted code (imperative code). In the same line of thought, to overcome the limits of navigation in IDEs, Hunter [80] is a tool for the visualization of JavaScript applications. It provides a set of coordinated views that includes a node-link diagram that depicts the dependencies among the components of a system, and a treemap that helps programmers orientate when navigating its structure.

We used a different strategy to bring related chunks of code closer by inserting "remote" code into the currently edited one. Similarly, "code portals" [42] embed various types of context information uniformly into the main source code view in proximity to the relevant source code. "Fluid source code views" [78] consists of insertions of relevant remote lines of code in the edited file. The aim is to provide the programmer with the control and data flow directly in the edited code, thereby minimizing navigation. However, it is applied to object-oriented programming and relies on displaying hierarchies of methods. SimplyHover [127] is a plug-in for Eclipse that brings the "if" condition next to its "else" counterpart. Theseus is an IDE extension that visualizes the run-time behavior of a program within a code editor by displaying real-time information about how the code actually behaves during execution [155, 156]. It provides the programmer with the number of calls of a particular function and a collapsible tree of calls. Like Causette, Theseus augments the edited textual code by displaying the number of calls to the left of the function header. By contrast, we target another type of information: the causal activation chain. We also focused on state activations in an FSM but used a quantitative representation of activation recency instead of a numerical representation of the number of calls. We took the Theseus evaluation method as inspiration for the evaluation of Causette.

5.1.3 Code representation and animation

Causette builds upon previous work about graphical code representations. For example, InterState [218] is a programming language and environment that supports developers in writing and reusing user interface code.

InterState mixes texts and graphics to represent interactive behaviors using a combination of FSMs and constraints. It also provides programmers with a visual notation to facilitate code navigation and understanding. SwingStates [10] makes clever use of Java anonymous inner classes to describe FSMs. However, the specification of the interactions is done at a local level, and SwingStates does not offer support for understanding causal chains.

From a more theoretical standpoint, source code shares much with a text to be read [237]. There is a distinction made in the literature between textual and visual information. But it is not clear-cut, and it is not obvious whether to favor the latter or the former. Visual programs can even be harder to read than textual programs [108]. The cognitive dimensions of notation is a framework that helps designers analyze interactive tools, including programming environments and languages, may they be textual or graphical [33]. The Physics of Notations framework focuses on the properties of graphical notations [197]. Unifying textual and visual languages shows that both types have much in common and that both should rely on the capability of the human visual system [67]. More recent work on code reading is relevant to Causette. There have been studies on how programmers read natural language text and code. Some results indicate that code reading is less linear [46, 220] than prose reading because programmer focus on the program execution flow or that code regularity (same structures repeated time after time) reduces code reading complexity [128]. The effect of ordering on comprehension has been studied [128], with a focus on the ordering of methods. It motivates design principles for Causette, in the sense that it invites to support the matching between the linearity of reading order and the readability of the execution flow.

According to Victor, an environment and language suitable for programming should allow one to "follow the flow" and "see the state" [281]. The author designed several interaction techniques and representations to support those two concerns. However, they involved following an imperative flow and data states, while Causette targets the interaction flow or the interactive state. The author claims that the features of the environment are less important than the particular ways of thinking they support. We strive to do this for interaction programming: provide programmers with interactions and representations to better apprehend causal relationships.

To augment and represent textual code efficiently, we used animations. Text animations help understand changes in information display [61]. Gliimpse [85] or Diffamation [60] share with our work the use of animations in code: they offer animated transitions to different parts of a text (latex markup code and rendered document in Gliimpse, revision history of textual documents in Diffamation). Gliimpse allows users to check and navigate the code without leaving the text editor. However, Gliimpse animates from code to rendering and not from code to code. Finally, animations of word-scale graphics within texts also enable to follow the rearrangement of graphics and make them easier to compare thanks to a vertical alignment [101].

5.2 Informal ideation with our 12 interviewees

The last 15 minutes of the interviews presented in Chapter 1 consisted of letting the interviewees imagine and describe a solution to be integrated within an IDE. We shortly sum up the three major solutions that emerged, common to several participants.

The first recurring solution was providing a *"trace"*, a *"kind of chain"* or the *"dependencies between files"* within the editor, at the programmer's fingertips (within the edited

file). I12 for example imagined the following: *"I don't want to leave the file, I'd like something that allows you to trace, to give me the dependencies between files. At the very least, I have another window or another column where I could walk through these dependencies. (...) Because I'm actually a developer with a very low abstraction capacity, I learned development through sequential development, so I like it when it happens in order. "* In the same line of thought, I10 and I11 wanted something alike: *"I could have on the right information about where the variable appears, where it is defined, used, redefined? Because here in the IDE you have some stuff available to help, of course, but it forces you to scroll up and down in your file and wander around, it's boring."*, *"It would be really important to have a tool that allows you to trace all these interconnections and it would also be nice to be able to comment on them. To say, 'Here's something happening, be careful, because I'm modifying an interconnection here'. Even for collaboration it would be really important"*. I9 evoked what she liked in search tools available in *Unity* and suggested how they could be completed: *"So on Unity, there was something pretty good. For example, you move your mouse over main_player and you do control, click and it sends you back to the window where the component is created, then you can click again and it sends you back to the other one. And in the end it allows you to better understand what happened, to make the links. It makes a kind of chain. But I would like to see it in a single glimpse, without having to click and navigate"*.

The second type of recurring solution focused on the improvement of FSM coding. Participant I5 wanted *"to see the current state and see that it's the correct one. That would help me too. And also that it's colorful. Everything that goes to the "radarstate" should be colored, for example. And it would be nice to dynamically rearrange them. Let me see everything that goes to "radarstate"*. (I5).

Finally, participants thought of solutions for supporting animation settings and frequency adjustments of received signals or data (I5, I11, I12). I5, I11, and I12 described sliders and graphs that could be displayed within the IDE and where the programmer could select values and see them automatically adjusted in code. Since the last idea was very specific and could not serve as a guiding principle for an IDE design, we decided to leave it aside and consider it later as a supplementary tool to be integrated.

5.3 Requirements and Design principles

We devised from the previous literature survey and insights from Chapter 1 three requirements for designing interaction techniques that would support a programmer in understanding interaction code. We aslo devised these design principles to fulfill the requirements.

Most design principles and interaction techniques leverage a textual code editor. Even if some graphical representations can be used to represent data-flow or FSMs, text-based editors are still heavily used as they provide features deemed usable by many programmers. Still, we hypothesize that the regular, mostly 1D, textual presentation of

causal relationship constructs spread across multiple files does not help programmers to understand the causal chain.

We concretely demonstrate the interaction techniques through use cases.

5.3.1 Requirements

The understanding of interaction code poses specific challenges arising from the multiplicity of causal chains and multiplicity of files splitting the description of behaviors. We wanted to encourage solutions available directly in the edited source code file, saving navigation time across files.

Hence our interaction techniques should support programmers in:

- Understanding the causal dependencies [**ReqCausal**]
- Backtracking the origin of a causal propagation, overcoming the inconvenience of code split across files [**ReqNoSplit**]
- Dynamically visualizing transitions and state activations [**ReqDynamicCaus**]

To the best of our knowledge, related work indicates that although code lines integration within edited source code is not new, these technique have not been applied to interaction code and causal relationships understanding.

5.3.2 Design principles

The first design principle that we followed to fulfill [ReqCausal] is to enable the programmer to make the y-ordering of lines of code consistent with the expected execution ordering [**DgnYRearrange**]. Reordering is especially important when the programmer wants to apprehend the multiplicity of execution paths due to uncontrolled sequences of external events.

The second design principle we followed to fulfill [ReqNoSplit] is to bring together the causal relationships that are far away from each other in the source code, including in the same file [**DgnTogether**]. This is especially important when coping with causal relationships in several code locations.

The third design principle is to take advantage of the text representation and the properties of visual variables from *Semiology of Graphics* [31] to understand the code and its execution. Notably, the first three interactions reorder the programming constructs to display them like an imperative, y-ordered control-flow. Note that the first design principle can be considered a special case of the third one, but its importance deserves a proper principle [**DgnVisVar**].

Finally, the fourth design principle consists in using animations to help the user apprehend the changes of the representations **[DgnAnim]** [61, 255], especially for [ReqDynamicCaus]. We also considered the guidelines related to the animation of lists, as lines of codes can be considered lists [255].

To the best of our knowledge, **[DgnYRearrange]** and [DgnTogether] have never been identified and used in past work. [DgnAnim] is not new but has not been applied to code-to-code transformation. [DgnVisVar] is not new [67] but has never been applied to interaction code.

Though following the same design principles, the interaction techniques were not designed to be entirely consistent: our goal was to explore the design space and how well the techniques would support program comprehension. Currently, the interactions are implemented in a GUI that enables a programmer to launch a Smala program and explore its source code with the interactions. Even though the first three interactions may be available statically (e.g., without running the explored source code), our implementation relies on the run-time *initialization* phase of the tree of processes and the reflexive capabilities of the Smala execution engine. Only the fourth interaction relies on the run-time *execution*. Relying on the execution engine enabled us to prototype the interactions without implementing a static analyzer.

5.4 Interactions

In this section, we present four interaction techniques through use-cases. The use-cases involve programs written in the Smala language. We will see in Section 5.7 that the interaction may also apply to other programming languages that share with Smala the same concerns. With Java or C++/Python/Qt programs, the entire causal chain specification can be spread over several files, making it impossible to visualize and difficult to understand. The Smala FSM also exhibits some scaling issues with large numbers of states and transitions, and with nested FSMs. The same issue also arises in other languages. A video presentation is available [3].

5.4.1 Interaction 1: reordering a data-flow

The first interaction technique provides the programmer with means to navigate inside a data-flow and display it like an imperative control-flow. It addresses in particular the issues and cases mentioned by the interviewees, as labeled by the tags 'Event and order' in Tables 1.3 of Chapter 1. The two use cases in the following are inspired by bugs described by our interviewees (I1, I3, I5, I7, I9, I11), when trying to understand why a piece of information is not updated on a Graphical User Interface. The resolution of these bugs, as described by the interviewees (see Table 1.2)involve in particular asking what triggers an event or propagates an activation.

[3] `https://www.youtube.com/watch?v=QjcB9ZaOxL8`

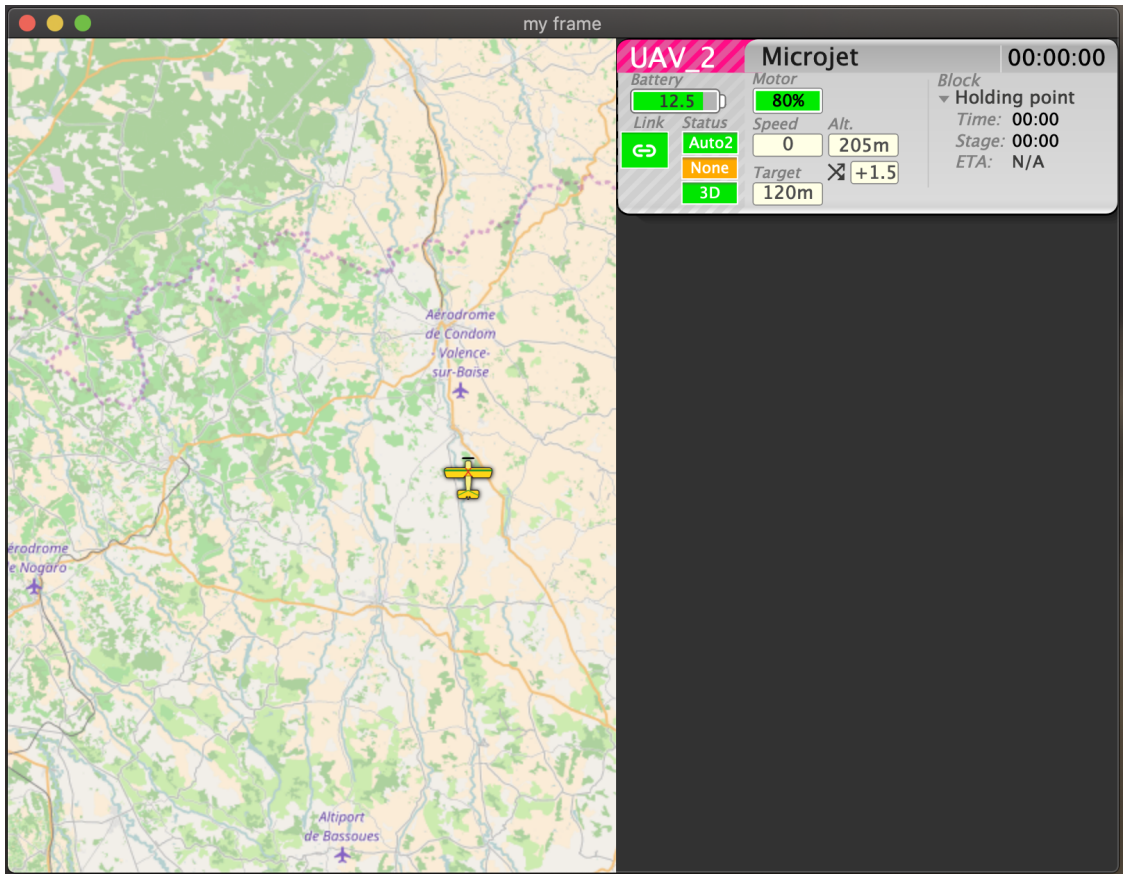

Figure 5.1: Drone ground station application, as used in the first use case for *Interaction 1*

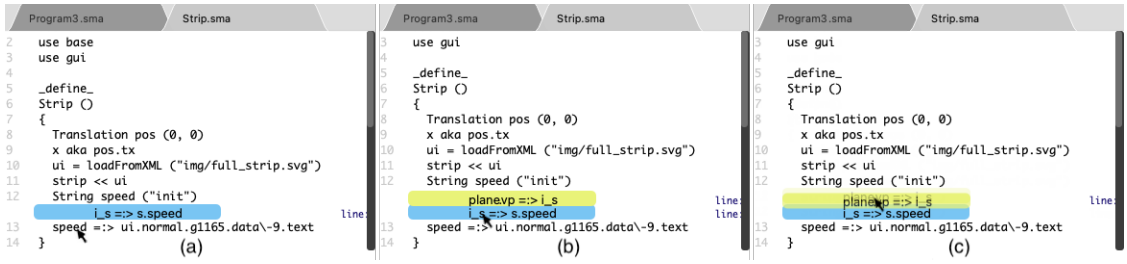

Figure 5.2: *Interaction 1.* Reordering a data-flow: (a) the user clicks on the left of a connector, an animation inserts the upstream data-flow construct (b) upon clicking on the new line, an animation inserts the following upstream construct (c) the user clicks on the new line, the system animates the line up-and-down to specify that the beginning of the data-flow has been reached.

5.4.1.1 Use Case 1: Spotting a broken data flow

The first use case is as follows. We use here the example of a prototype for the drone control user interface, as showed in Figure 5.1. The programmer of that drone ground station application is faced with unexpected behavior while testing interaction code. Some text in the graphical interface is not updated as it should be, which is likely the symptom of a broken data-flow. She wants to trace the code to identify the cause. The drone interface relies on the connection between parsed information from a software bus to their graphical display within a so-called "strip" on the right panel. In this example, the cause to be identified is a missing connector: the speed of the drone model displayed in the view is not connected to the corresponding data from the bus.

We designed an interaction technique to navigate the data-flow (Figure 5.2): clicking on a variable on the left-hand side of a data-flow construct [4] summons an animated apparition of the upstream construct connected to the said variable. The upstream construct appears in a line of code just above the clicked construct. Recursively, the programmer can click on the summoned lines to display further upstream constructs. If multiple sources are connected on a property, they are all shown using a line each. If nothing is connected, a short animation quickly bounces the clicked line up and down. This signals to the programmer that the origin of the data-flow has been reached. The summoned lines may come from the current text file being edited but also from other files. In this case, the source filename is appended at the right of the line as a hyperlink that enables the programmer to jump to the code.

Step by step, the programmer is thus able to trace back the absence of a causal construct that should have led to the activation of the line of interest. In the example, the user cannot see anything connected to the property named *plane.vp*. This cannot be automatically identified as a mistake since such behavior could have been legitimate in some applications (in a program, many properties are partially connected).

The interaction also applies to downstream constructs by clicking on the right side of the arrow or the declaration of a property. It is also available for transitions in FSMs: one can explore the chain that leads to or depart from a transition event.

Interaction 1 relies on the three design principles to help understand the data-flow:

- Bringing together related causal relationships located in the same file or in different files [**DgnTogether**],
- Making the y-ordering lines consistent with the order of the data-flow [**DgnYArrange**],
- Using visual variables (here the planar variables x and y), used selectively and orderly [**DgnVisVar**].

[4] In Smala a data-flow is expressed as $a =:> b$, which means that the value of a is copied into b each time the value of a changes.

The insertion above the lines being read by the programmer keeps the context in which the programmer tries to understand the code. It also displays the lines of code as if they were next to each other. In that respect, it differs from a "Jump to Reference" command in a traditional code editor, which completely changes the content of the window or, at best, provides a transient pop-up on top of the currently edited window.

It is important to note that reordering would only concern the appearance of the program, not its actual source code, and it would not change its semantics. The representations of code change upon users' request (to explore the downstream and upstream causal chain) should also be set back to the original arrangement — upon request. Hence, the quality of the software system's design would, at worst, be retained.

The sequence of lines of code obtained, perceptually ordered in the y-dimension and x-aligned [67], is reminiscent of the sequence of lines of code in an imperative language. However, here the y-dimension of the source code is mapped to the causal relationships (reactive language) instead of the program counter (imperative language). At the same time, the x-alignment specifies that all lines belong to the same data-flow (reactive) instead of control-flow (imperative). It makes the causal chain directly visible.

5.4.1.2 Use Case 2: Spotting a wrong link

The same interaction technique makes it easier to spot a wrong link in the data-flow: the programmer can quickly identify a process wrongly connected to another one. In Figure 5.3, the programmer has unfolded the causal sequence starting from line 13. After three steps in the causal chain, the code shows that "speed" has two sources at that level: the property *plane.vp* is erroneously connected twice to two different parsed information from the bus. Again, this cannot be automatically identified as a mistake, since such behavior could have been perfectly legitimate in some applications. The use of a dedicated color for each level of source in the hierarchy facilitates the identification of this kind of error [DgnVisVar]. If there are several lines with the same color, it means that several flows feed the same property.

5.4.2 Interaction 2: reordering *textual* FSMs

We derived the third use-case from the interviews with FSMs. The use case is about the programming and debugging of FSMs: when one needs to reread her or his own code and check which transitions lead to a particular state. In Table 1.2, the recurrent issues that inspired Interaction 2 are the following: "checking what event triggers a transition", figuring out what the allowed transitions are". In the example in Figure 5.4, the FSM has 7 states and 20 transitions. It is thus difficult to apprehend the whole FSM code and understand its behavior given a particular sequence of events. The use-case is as follows: there is a suspicious, transient activation of the *checkLoopState* state. The user wants to understand the causal events that lead to this state, what the state is activating in turn, and what causes its exiting. However, the current textual state of the FSM

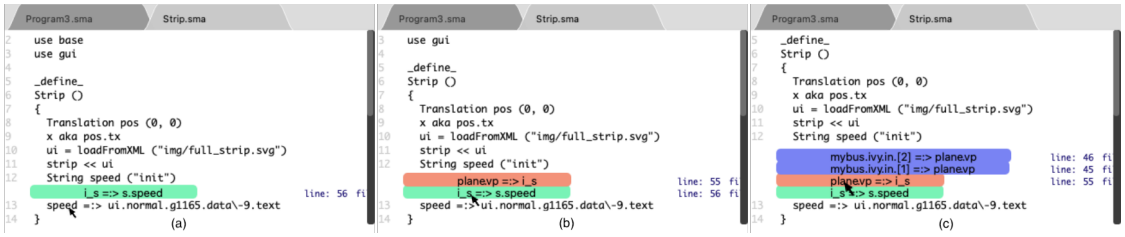

Figure 5.3: *Interaction 1*: navigating a data-flow: (a) the user clicks on the left-hand side of a connector (b) an animation inserts the immediate upstream data-flow construct. The source is from another file (c). The user clicks on the new line, and an animation inserts the following upstream construct. At that depth, multiple sources are connected to the same variable.

representation makes it difficult to visualize such a sequence as it forces the user to look for the involved transitions and to hop from one line to another in arbitrary directions.

We designed an interaction technique based on reordering an FSM's elements. The programmer can hover over a state, highlighting with a green background the transitions that go into or leave the state. S/he can then click on the state and see a smooth, animated change of the layout of the transitions around the clicked state to make 'in' transitions lie above the state and 'out' transitions lie below the state. She can continue exploring the follow-up causal chains by clicking on another state and seeing the 'in' and 'out' transitions move around it.

Interaction 2 relies on the same design principles as Interaction 1, but is applied to the causal chain related to FSMs (with states, transitions, and events) instead of data-flows (with variables and operators such as connectors, bindings, or assignments). Similarly, the resulting sequence of lines of code, ordered in the y-dimension and x-aligned, is reminiscent of the sequence of lines of code in an imperative language and allows a developer better to understand the control-flow and its associated causal chain. In addition, the animation allows the programmer to catch a glimpse of which transitions take the system to a specific state, and which transitions make the system leave the state in question.

Navigation within FSMs and dataflows can be combined. One can rearrange transitions around a state (interaction 2), and click on the process `to_check_loop` that triggers a transition... (interaction 1):

```
check_state -> checkLoopState(to_check_loop)
```

... to summon the apparition of an upstream binding:

```
checkSound.t.end -> to_check_loop

check_state-> checkLoopState(to_check_loop)
```

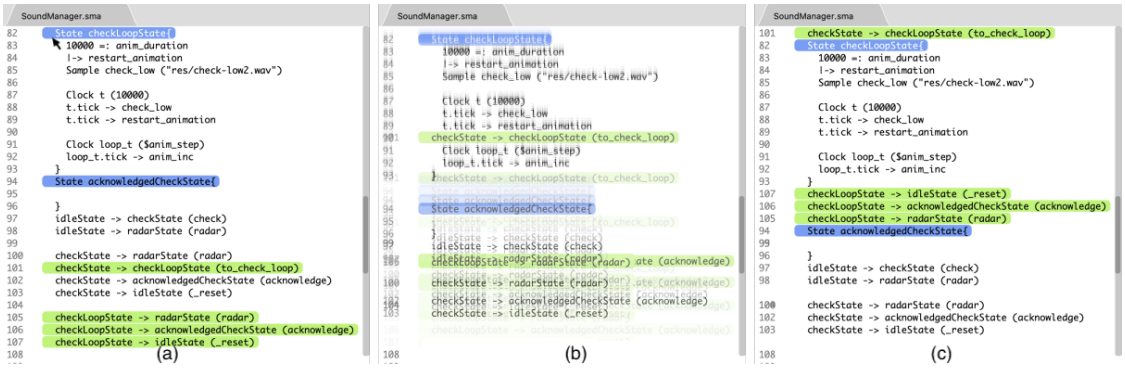

Figure 5.4: *Interaction 2.* Reordering FSMs: (a) The user hovers over on a declaration of a state (blue) which highlights the in and out transitions (green); (b) the user clicks on the state, and the system animates a rearrangement of the transitions; (c) the new y-ordering of the statements matches the causal relationships being analyzed (in-transitions above state above out-transitions).

5.4.3 Interaction 3: reordering *graphical* FSMs

The previous interaction techniques demonstrate how text-based representations of interaction code could be re-arranged to follow the control-flow better. The use-case from Interaction 2 was concerned with the understanding of the control-flow "around" one particular state. Here the use-case is extended to multiple states.

A popular representation of FSMs relies on circles depicting states and arrows depicting transitions, annotated with the event that fires a transition. Even if the two representations ("textual code" and "circle-arrow") seem different, we can smoothly transition from one to another to adapt the view according to the ongoing task.

We designed an interaction technique that rearranges the circle-arrow representation (Figure 5.5)[5]. Starting from the circle-and-arrow representation, the user draws a line that passes through states, transitions, and events. The users should specify their gestures according to the sequence of events they want to analyze. After the gesture has been performed, the system animates a rearrangement of the circle-arrow representation. The final arrangement allows the programmer to read from top to bottom the causal chain involving a succession of states and events.

Again, such a representation provides the programmer with a sequential reading of the code, which makes causal ordering salient. Remarkably, the final circle-arrow representation is similar to the textual one. This supports the hypothesis that graphical and textual are not so different [67]): the visual representation reuses the assets of the textual one, such as selectivity of the x-dimension (i.e., left alignment [67]) and the ordered perception of the y-dimension to depict the order of the control-flow.

[5]this interaction has not been implemented and is a prototype to be explored in future work

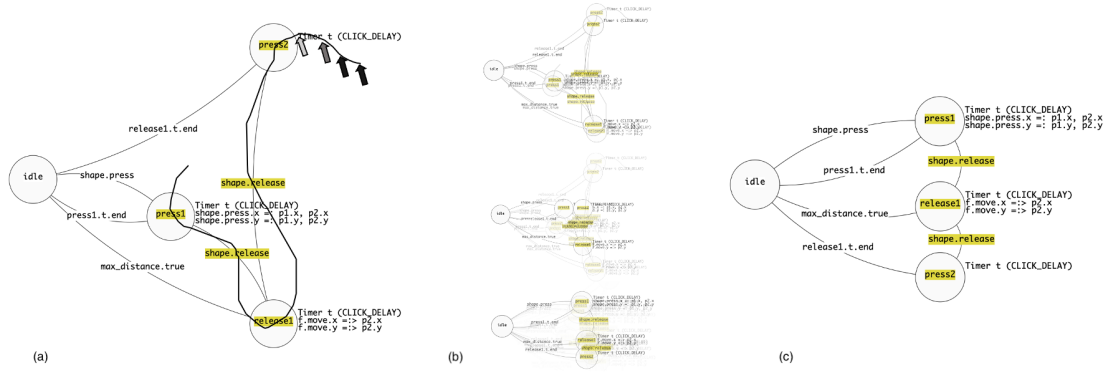

Figure 5.5: *Interaction 3.* Reordering an FSM given a path selection: (a) starting from a canonical graphical circle-arrow representation of an FSM, the user draws a path through states and transitions according to the causal chain she wants to explore; (b) the system reorders and animates the circles and arrows; (c) the final layout is similar to the corresponding text representation.

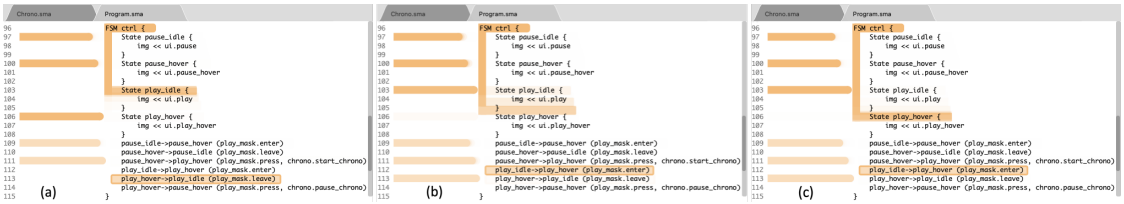

Figure 5.6: *Interaction 4.* Showing in the text editor the activation of states and transitions during execution. (a) The FSM enters state *play idle*, due to activation of the highlighted transition (b) The selected state moves to *play hover* (c) State *play hover* is highlighted and changes in the history sidebars are updated.

5.4.4 Interaction 4: showing the dynamics of FSMs

The dynamic behavior of FSMs can be hard to understand. This refers to other issues mentioned by the interviewees about FSMs, like "Checking which state the system is in". The use case is a follows: on a GUI lies a displayed clock, coded with a FSM, which should switch from clock to timer mode upon request on a "play" button. However, nothing happens when the button is clicked. The programmer wants to check whether the issue is a faulty transition of the FSM.

We designed an animation that highlights the activation of states and transitions during execution (see Figure 5.6). Each time an FSM enters some state, the state is highlighted, and the transition that activated it is both highlighted and outlined. States might be declared remotely from the declaration of the FSM itself, especially in an embedded FSM. Therefore, the representation also connects the activated state (e.g., *play_hover*) to the

parent FSM (e.g.,*ctrl*) in which it is declared. As time goes by during execution, the previous activated states and transitions are denoted by horizontal sidebars that progressively fade away and shrink towards the left (see Figure 5.6), much like a vertical VU-meter in music players. The progressive fading gives the user a sense of the history of activations. This representation also shows whether a state or a transition has been activated at least once. This is particularly useful when the transition between states is very fast. Compared to Theseus [155, 156] (presented previously in Section 5.1.2), we think Interaction 4 is better suited to interaction programming, especially when activations are fast and when one tries to figure out the order of activation (by comparing bar lengths).

5.5 The Smala language and Causette's implementation

We developed Causette in Smala. For the sake of replication, we describe here the strategies we used to implement Causette. Smala is a textual, interaction-oriented programming language dedicated to developing highly interactive software. It takes inspiration from classical reactive languages such as Lustre [53] or Esterel [27, 28, 92], adding notably a specific syntax and a smooth integration of graphical content.

At the conceptual level, a Smala program is the specification of a dependency graph of coupled nodes going from a set of event sources to a set of output nodes or sinks. Such a declarative specification is akin to the way causal chains are declared in an imperative language/framework such as Java or Qt: adding listeners (java) or connecting signals to slots (Qt) is a way to *declare* a causal chain during the initialization phase of an interactive program. The execution flow is triggered by the occurrence of events propagated by an activation vector, resulting from sorting the dependency graph. Smala's syntax is described in Appendix A.

The Smala language is built upon a set of C++ libraries named Djnn [6] that provide an implementation of the Smala nodes and an execution engine. This execution engine has some useful features that helped us implement our interactions. In particular, all nodes in a Smala program have an activation state to which it is possible to subscribe. This enables the dynamic visualization of the activation status of any node such as the states and transitions of a FSM, as in 5.4.3 and 5.4.4, using a simple subscribe pattern. Causette takes a running tree of Smala nodes as a parameter, browses it and subscribes to the various FSM states and transitions. Moreover, when compiled in debug mode, the C++ nodes keep a reference to the Smala file and line where they have been instantiated. The application knows which part of the code must be animated when a node is (de-)activated.

The animation for the data-flow is slightly more complex. We modified the Djnn libraries to ensure a property node involved in a data-flow keeps a reference to all properties

[6] `https://github.com/lii-enac/djnn-cpp`

connected. Combined with the previous one, this additional reflexive feature allows us to display the successive steps of a data-flow easily for Interaction 1 and 2. The drawback is increased memory consumption and more management, but this concerns only debug code, not production code.

5.6 Evaluation

We conducted a study on the usability of Causette for programmers of interactive systems faced with the typical problems identified through the interviews (see Chapter 1). We wanted to assess how much more usable Causette would be compared to a traditional text editor. Due to the nature of our research (facilitating some program understanding tasks), we expected that it would be difficult to design a controlled experiment that would be both statistically and practically significant. We expected that measuring the completion time of a program understanding task would be highly dependent on inter-individual differences. We thus followed the principles of Single-Subject Research [198]. Single-Subject Research involves testing a small number of participants and focusing intensively on the behavior of each individual (as opposed to, e.g., means of measures across groups), and measuring strong and consistent effects that have biological or social importance [198]. In the following, we present the results per-subject instead of aggregated measures of multiple subjects. To design the experiment and report on its results, we followed the principles of Fair Statistical Communication in HCI [83]. In particular, when suitable, we asked quantitative research questions and stated the effect size.

5.6.1 Research questions

We focused on the following research questions:

RQ1. How much would Causette facilitate understanding causal structures in interaction code?

RQ2. How much would Causette make a difference when given a complex "interactive bug"?

RQ3. How much would programmers benefit from inserting related code lines into the source code being analyzed compared to common methods in text editors (e.g. "find", "jump to definition")?

5.6.2 Participants

We recruited 10 participants. All of them were men (more on this in section 5.7) and were of age 20-25, 1 of age 25-30, 4 of age 30-40. They had at least an M.S in Computer

	Age	Gender	Training	Smala use pre-experiment	IDE & debugging habits
S1	30-40	M	Ph.D in Computer Science	min. 6 months	Sublime, Visual Studio
S2	20-25	M	M.S. in HCI	min 4 months	Sublime, Eclipse
S3	20-25	M	M.S. in HCI	min 6 months	Sublime, Eclipse, Visual Studio
S4	20-25	M	M.S. in Computer Science	min. 4 months	Sublime, Eclipse
S5	20-25	M	M.S. in Computer Science	min. 6 months	Sublime
S6	30-40	M	Ph.D. in HCI	min. 3 months	Sublime, Visual Studio
S7	30-40	M	Ph.D. in HCI	min. 1 year	Sublime, Visual Studio
S8	20-25	M	M.S. in Computer Science	min. 4 months	Sublime, Eclipse
S9	30-40	M	M.S. in HCI	min. 4 months	Snoop, Sublime, Visual Studio
S10	25-30	M	M.S. in HCI	min. 1 year	Sublime, Eclipse

Table 5.1: Demographic data for the lab experience

Science or HCI, and 3 hold a Ph.D. degree. Half of them had more than three years of professional experience in interaction programming with Qt, Javascript, or Swing. They were proficient in Smala and were using Sublime Text as an IDE. Sublime Text is a typical textual code editor with features such as those mentioned in RQ3, also available in Microsoft Visual Studio Code.

5.6.3 Experimental design

We are aware of the challenges arising in the design of code comprehension tasks, and existing work has helped us analyze the limits of our evaluation study [91]. Our experimental design is inspired by two works that evaluate new features for IDEs, Theseus [156] and CodeBubbles [41]. We thus conceived a set of tasks related to code comprehension and debugging. We used the code of a real-size application actually in use in Air Traffic Control (ATC), as displayed in Figure 5.7. It consists of a GUI that supports Air Traffic Controllers in detecting a conflict between two flights. It displays information about each flight in a sector in the form of a strip e.g., the callsign of the flight, its altitude, and details about its route. The source code amounts to approximately 1,000 lines of Smala spread across 7 files.

Participants were given 6 tasks to complete in total (A to F), those 6 tasks falling into 3 categories (dataflow, value, debug). We designed the tasks within a category to be equivalent in terms of difficulty. The tasks are inspired by actual cases described in the interviews [7].

- **Dataflow comprehension questions (A-B):** 2 tasks targeted the understanding of dataflow and required the participants to identify what could trigger the activation of a variable or what the variable could activate down the causal chain. In other words: "What causes process A and what can A cause?".

  A. Tracing the destinations The task consists in finding all the possible destinations of a variable named *critical_time*. In other words: what other variables could *critical_time* activate, possibly with value propagation?

[7] Note that the study's participants are not our interviewees.

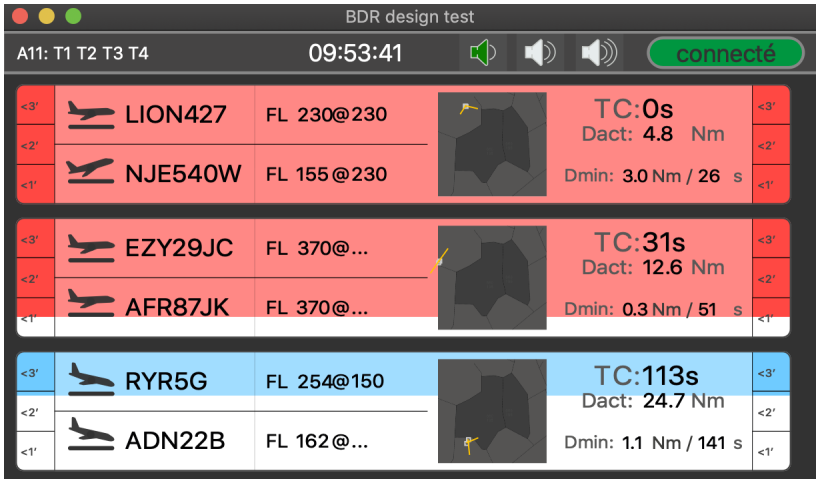

Figure 5.7: Air Traffic Control app used for the evaluation

B. Tracing the sources of an event The participant was asked to find all the possible sources that could trigger a *reset* event in an FSM.

- **Value questions (C-D):** 2 tasks consisted in finding the values a variable could take.

  C. String variable values: Participants were to find all the possible values a string variable named *status* could take.

  D. Integer variable values: Similarly, participants had to find all the possible values an integer variable named *soundLevel* could take.

- **Complex comprehension and debugging tasks (E-F):** Finally, 2 tasks consisted in fixing complex bug. The tasks require a deeper understanding of the overall behavior of the program. Although E and F targeted either FSM or dataflow issue, we consider them as equivalent in terms of causal complexity. Given the fact that all our participants had at least two months experience in Smala, we assumed they were equally prepared to FSM and dataflow issues.

  E. Missing textual object: The task involved solving a dataflow problem, where the value of the flight callsign, originating from a parsed piece of information from a software bus, was not correctly connected to the textual property of the GUI. The error was a property naming problem due to a former incorrect *copy/paste*. The problem was perceivable on the GUI: on the displayed strip, two *callsigns* should normally be displayed, but only one appeared.

  F. Faulty alarm: The participant was asked to debug an FSM in charge of the sound management of the application. That FSM had an embedded FSM, with 20 transitions and 9 states. When correctly coded to match the expected behavior, the system was supposed to work as follows. Four different audio alarms alert the Air Traffic Controllers about the ongoing conflicts, as long as the system received

data streamed from the software bus. If the software bus stops streaming data (e.g. because of a network problem), after a few seconds, a message is displayed ("no data received"), and the auditory alarms shut down automatically. We introduced an error in the code by deleting one transition of the FSM that should have reset the application in case of streaming abortion. Consequently, the FSM ended up stuck in some state $RadarLoopState$, and an alarm kept ringing, no matter whether the software bus was streaming or not. Participants had to figure that out and fix it.

Figure 5.8 is a summary of the experience design.

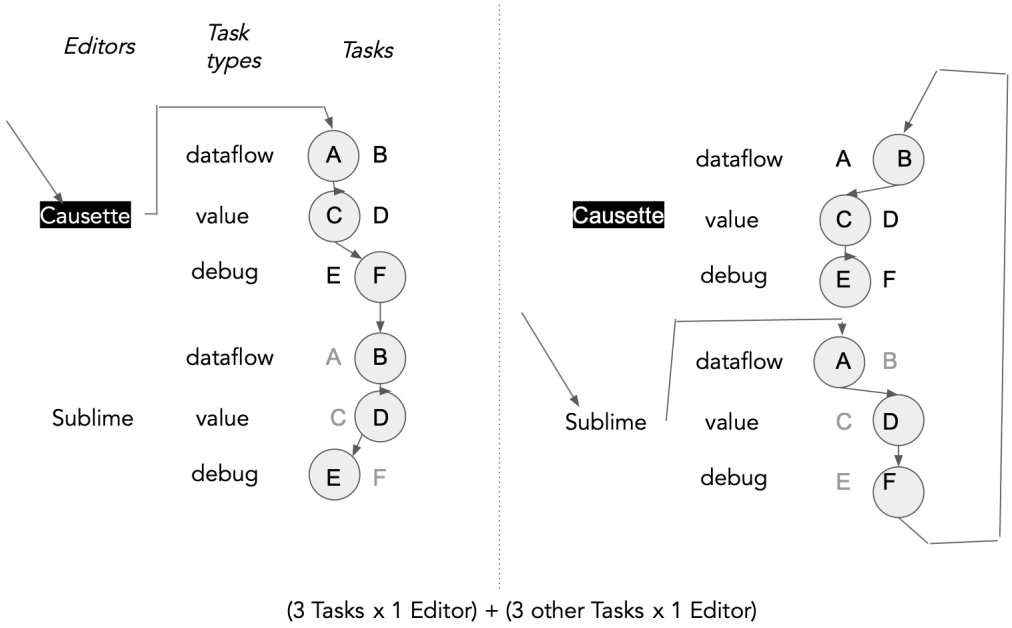

Figure 5.8: Summary of the experience design

There were 2 conditions: a control condition with the participants' usual editor (here Sublime), and a condition with Causette. Sublime is not a full-blown IDE, and we discuss why it was chosen for the control condition in section 5.7. To facilitate within-subjects comparison, each participant was assigned three (from each category) tasks in the control condition and three tasks using Causette. To counterbalance an order effect, half of the subjects completed all of their control tasks first, while the other half completed all of their Causette tasks first. The selection of the tasks across categories was randomized.

Before the evaluation, each participant was given a 10 minutes summary of the code: an overview of the architecture and the functions of the app, and the ATC vocabulary to understand identifiers in the code. The code itself did not include any comments. Once the participant had read the question and was ready to perform the task, a timer was launched. All the interaction techniques were available for each task. Up to seven minutes were allowed for each of the A-B and C-D tasks, while fifteen minutes were allowed for

E-F tasks. When participants felt they completed the tasks, they told us, and we stopped the task, recorded the duration regardless of the correctness of the answer, and asked for a confidence rate. Participants were also able to give up a task if they felt they could not perform it.

If the timer reached the maximum allowed time, we interrupted the task. In total, up to one hour was devoted to the tasks. At the end of the experiment, we made the participants fill out a System Usability Scale (SUS) [43] questionnaire on Causette, followed by a 30 minutes interview to get qualitative feedback.

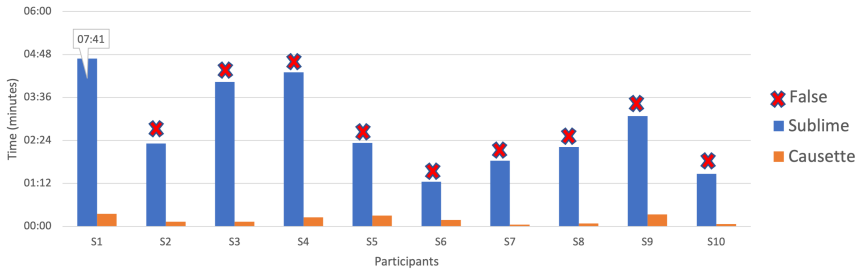

Figure 5.9: Completion time for the **dataflow tasks, comparing control condition (Sublime) and Causette condition. The absence of a red cross means the answer is correct.**

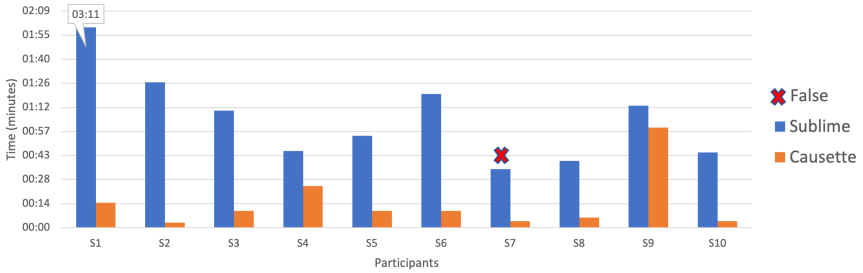

Figure 5.10: Completion time for the **debugging tasks**, comparing control condition (Sublime) and Causette condition.

5.6.4 Results

The complete results of the study include completion time, completion success, confidence rate and SUS score. They are available in Table 5.3 in the appendix). Figures 5.9, 5.10, 5.11 shows the completion time and completion success per subject for each type of tasks and according to the two conditions.

The results of the study are summarized in Table 5.2.

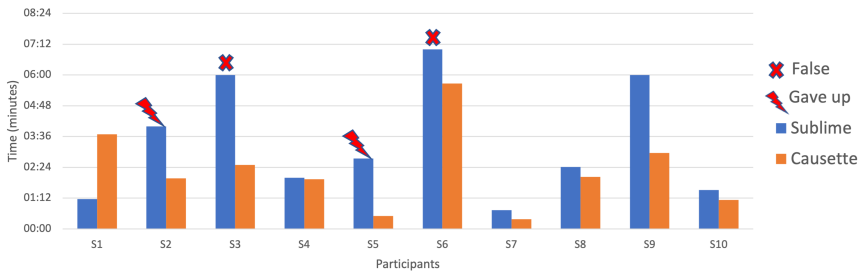

Figure 5.11: Completion time for the **third category of tasks (debugging problems)**, comparing control condition (Sublime) and Causette condition.

	Type of task						SUS score
	Dataflow question		Value question		Debugging task		
	A	B	C	D	E	F	
S1	✓	✓	✓	✓	✓	✓	70
S2	✓	x	✓	✓	✓	. (gave up)	92,5
S3	✓	x	✓	✓	x	✓	92,5
S4	x	✓	✓	✓	✓	✓	92,5
S5	x	✓	✓	✓	✓	. (gave up)	95
S6	x	✓	✓	✓	x	✓	62,5
S7	✓	x	✓	✓	✓	✓	77,5
S8	✓	x	x	✓	✓	✓	95
S9	x	✓	✓	✓	✓	✓	92,5
S10	x	✓	✓	✓	✓	✓	95

Table 5.2: Task success and usability score. The symbol ✓ indicates the participant gave the right answer. A cross indicates the answer was incorrect or incomplete. Two participants gave up one task each. The blue cells show that the task was performed with Causette

5.6.4.1 Effectiveness - Degree of achievement

The participants could all correctly complete the tasks in the Causette condition. This contrasts with the completion rate in the Control condition, where many participants could not provide a correct answer or gave up. In particular, 9 out of 10 performed the dataflow tasks partially correctly, as their answers were not exhaustive: they did not mention all paths of the dataflow, but remarkably, all but S8 felt confident.

Except for S1, each participant at least made an error or gave up in the Control condition. 4 participants could not complete the debugging tasks (2 gave incorrect answers, and 2 gave up). This suggests that users are not completely effective with a traditional code editor for these tasks, supporting our analysis of problems encountered by programmers. This also suggests that our interaction techniques make users more effective than

		Type of task						SUS score
		Dataflow question		Value question		Debugging task		
		A	B	C	D	E	F	
S1	Task success	✓	✓	✓	✓	✓	✓	70
	Response time	07:41	0:21	3:11	0:15	1:10	03:41	
	Confidence rate	4	5	4	5	4	5	
S2	Task success	✓	x	✓	✓	✓	.gave up	92,5
	Response time	0:08	02:19	0:03	01:27	01:58	(04:00)	
	Confidence rate	5	4	5	5	5	5	
S3	Task success	✓	x	✓	✓	x	✓	92,5
	Response time	0:08	04:02	01:10	0:10	06:00	02:30	
	Confidence rate	5	4	5	5	1	5	
S4	Task success	x	✓	✓	✓	✓	✓	92,5
	Response time	04:18	0:15	0:46	0:25	1:56	02:00	
	Confidence rate	4	5	5	5	4	5	
S5	Task success	x	✓	✓	✓	✓	.gave up	95
	Response time	02:20	0:18	0:55	0:10	0:30	02:45	
	Confidence rate	3	5	5	5	5	1	
S6	Task success	x	✓	✓	✓	x	✓	62,5
	Response time	01:15	0:11	0:10	01:20	07:00	05:40	
	Confidence rate	4	5	5	5	1	1	
S7	Task success	✓	x	✓	✓	✓	✓	77,5
	Response time	0:03	01:50	0:04	0:35	0:44	0:23	
	Confidence rate	5	5	5	5	5	5	
S8	Task success	✓	x	x	✓	✓	✓	95
	Response time	0:05	02:13	0:40	0:06	02:02	02:25	
	Confidence rate	5	2	3	5	4	4	
S9	Task success	x	✓	✓	✓	✓	✓	92,5
	Response time	03:05	0:20	01:13	01:00	06:00	02:58	
	Confidence rate	4	5	5	4	5	4	
S10	Task success	x	✓	✓	✓	✓	✓	95
	Response time	01:28	0:04	0:04	0:45	01:08	01:31	
	Confidence rate	4	5	5	5	5	4	

Table 5.3: Experiment overall results: task success, response time, confidence rate and SUS score

a traditional code editor for all types of tasks.

Participants were always confident in their answers in the Causette condition. 4 of them (S3, S5, S6, S8) were not confident in some of their answers in the Control condition.

5.6.4.2 Efficiency - Time of completion

Figure 5.12 represents the time percentage gain per participant and task group. Except for S1 and S4 in debugging tasks, all subjects performed their tasks faster in Causette conditions than in Control conditions.

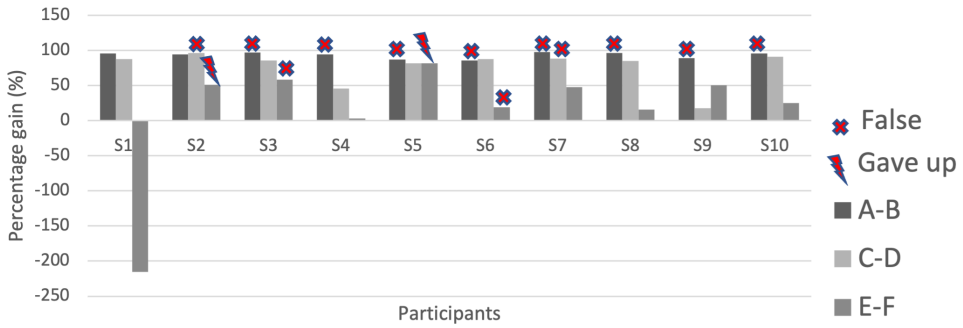

Figure 5.12: Time percentage gain per participant and per task groups when using Causette compared to Control (Sublime). We still indicate time of completion although the answer to the question or to the problem was false or if the task was abandoned.

In the dataflow tasks, users could perform 93% faster with Causette on average. Users could only answer partially in these tasks in Control condition, but we think the completion time is meaningful since they felt they had completed the tasks. They would have spent even more time had they resumed the tasks to find out all paths in the dataflow.

In the value tasks, users could perform 77% faster with Causette on average.

The results of the debugging tasks are more contrasted. As a reminder, in the control condition, S1 performed better, S4 performed equally, but two participants gave up and two gave incorrect answers. S1 is slower with Causette in the debugging tasks. S1 told us he was familiar with dataflow issues and less familiar with FSMs bugs, which could explain this outlier. S4 is as fast with Sublime as with Causette. S4 told us he had been focusing on FSMs in his code for two months before the experiment, which could explain his proficiency with Sublime at finding bugs. Except for S1 and S4, the 8 other participants performed faster with Causette. Ignoring incorrect answers, only five participants (S2, S3, S5 S7 and S9) performed significantly faster with Causette. S6, S8 and S10 performed faster with Causette but with a smaller margin.

All in all, these results tend to suggest that Causette makes programmers faster at fulfilling the dataflow and value tasks. It may make them faster at debugging with a smaller time gain for half of the participants.

5.6.4.3 Satisfaction

We gathered the SUS scores for Causette only. 7 out of 10 are higher than 90, and the remaining 3 (S1, S6, S7) range from 62,5 to 72,5. S1, S6, and S7 thought they might need external support to use Causette (score 3). S7 eventually stated that he would be able to learn to use it quickly. S1 and S6 did not feel very confident (score 3). S1 and S7 thought the system was not well integrated (score 3). S6 did not think he would use it frequently.

We present Causette as a set of interactions, and not as a system. If we polished the interactions as much as we could, the system itself that embeds them is still not very usable. We hypothesize that the comments of S1, S6 and S7 reflect the overall usability of the system more than the interactions per se.

5.6.4.4 Internal validity assessment

We examined whether the design of our experiment exhibits any bias concerning task equivalence or order (see Figures 5.13, 5.14).

The completion time between A-B, C-D, and E-F tasks in the two conditions tend to confirm that the two tasks within each category were of the same difficulty.

In the Causette condition, the order seems to have no effect. There could be an order effect on the completion time of the Sublime condition. It looks as if the use of Causette in the first place positively impacted the performances in the Control conditions for the data-flow and value tasks, while it is the opposite for the debugging tasks. However, the computed average times involve at most 3 values, even less when participants gave up or gave an incorrect answer. It is thus difficult to be conclusive or to comment.

5.6.4.5 Qualitative results

Because of the small number of participants, we ran a qualitative analysis of the results. After each session, we interviewed the participant to get qualitative feedback.

RQ1. How much would Causette facilitate understanding causal structures in interaction code?

Participants mentioned that they felt Causette saved them time to understand the dataflow (all the participants, except S6). The reordering of the in- and out- transitions was also described as time saving (S1, S4, S8, S10). S1 mentioned that he usually needs to draw FSMs when confused about the transitions between states. Participants S1, S8, S10 pointed out that it was interesting to display the current state of an FSM (in the real-time animation of the FSM) instead of printing it in the console.

S9 commented more in-depth: “You waste time, in general, going too deep when you don’t need to understand everything. You just need an overall idea of who’s activating

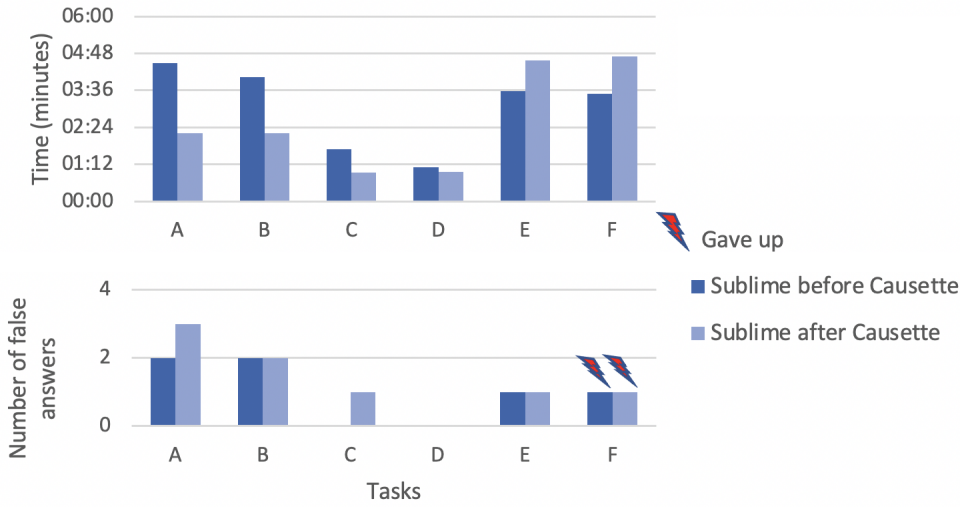

Figure 5.13: Condition ordering effect: completion time means (upper figure) and answer incorrectness (lower figure) per task in Sublime condition.

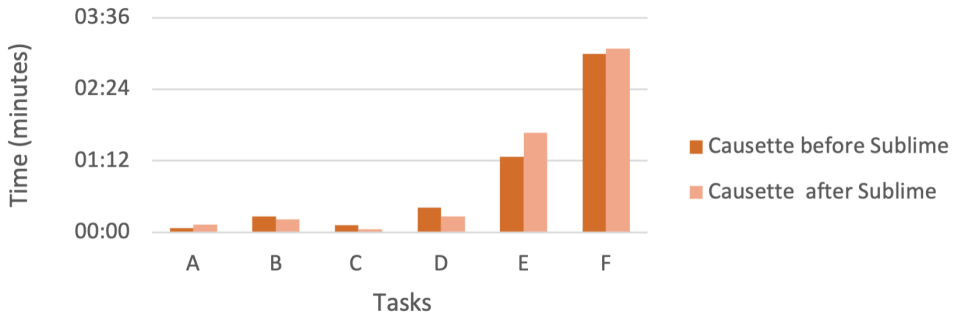

Figure 5.14: Condition ordering effect and completion time means per task in Causette condition.

who. With code for interfaces or systems like that, you have to do a lot of find tasks: "who gets what? Who triggers who?".

The issue about the declarative style of interaction code and how it prevents easy causal understanding was brought up: “Because what’s difficult is the declarative aspect, without logical links. You can write it in any order you want.” (S8). “you can often write in any order you want, but it becomes complicated to read back, and this is not just a problem in Smala (where you can write actions where you want) but in other languages too. I’m striving to put declaration code on one side and actions on the other. Otherwise you’re always jumping around.” (S9). S8 also developed an issue for FSMs: “with nested FSMs, after a moment I’m lost in the file, and having something that allows to put logical order between states and transitions, that helps.”

A few participants mentioned that the tool helped them construct or get quick access to

a “mental picture” of the causal relationships: “I don’t have to have a mind map all the time [...], there is no need to make drawings.” (S4). Another finding was that 6 out of 10 participants underlined what they felt would be the main use of the tool: to get into someone else’s code or one’s previous project. We heard the following comments: “very useful also to explain the code to other people” (S4), “real added value, especially when you read the code from someone else” (S7),

The animated insertions were described as presenting the causal relationships effectively. For the dataflow, participants mentioned they felt reassured being provided with a guaranteed, exhaustive list of sources and destinations (S1, S10, S3, S8, S4). S1 and S10 commented that with a traditional editor, they usually feel unsure whether they figured out the dataflow completely, and they took time to check that the property under study is not dependent on another file or is renamed somewhere else. S10 sums up this aspect as follows: “When I have a bug on an interaction, I need an exhaustive list of what interferes with it. Too often, there are hidden variables, and side effects. It’s hell. So it’s nice to have the guarantee with the tool that you have an exhaustive view of the flow.” (S10) This exhaustivity guarantee conveys confidence: “Having everything at hand guarantees that everything is there, we do not have the same confidence with a "find"” (S1). Such comments are consistent with the confidence rates we measured, which were (with one exception) equivalent to or higher with Causette than in the Control condition.

RQ2. How much would Causette make a difference when given a complex “interactive bug”?

Some participants commented that the two complex comprehension and debugging tasks were reminiscent of the problems they usually face. Several comparisons and equivalent use cases were provided. S3 said “it reminds me of many cases [...] where my code compiles and yet there is an error : I was creating my graphical objects, but they were aligned at the top left of my interface, and it took me time to understand that I had not initialized their length and height”.

In the debugging task, S5 said he could have completed the last task he gave up in the control condition, seeing clearly that the FSM was stuck in a state, but he got tangled up in staying focused on the wrong state. S4 and S7 elaborated on that aspect, saying the FSM animations are useful to locate which state the FSM is blocked in. They found the real-time animation and the history bar even more useful “to really see it”. S1, S4, S5, S6 found that reordering FSMs make transition errors more salient.

However, participants often mentioned the need to have a link from the GUI to the source editor, to be more efficient. Two participants mentioned *Snoop* [8], an existing tool they had appreciated in the past.

This had already been commented at length by a participant of our exploratory evaluation, and we develop that issue below in subsection 5.6.4.6.

RQ3. How much would programmers benefit from inserting related code lines into the analyzed source code?

[8] `https://github.com/snoopwpf/snoopwpf`

First, participants brought up that Causette avoids navigating at length: “with the tool we avoid unnecessary navigation. When you use diverse “find” functions, you often end up having too many occurrences” (S4). This could be supported by the difference in response time we found. Comparisons with search functions in different IDEs were often discussed by the participants, to the point where Causette was described as a “super fast search tool” (S5). S3 for example made a comparison with the Qt Creator: “you right-click to see the uses of the variable, but there is no difference between declaration and use, while here you really see where I come from, where I am initialized and where I go”. The alignment of process names on the x-axis when line insertion occurs was commented as useful by S3. It made the flow more salient. S2, S3, S4, S5, and S8 liked the simplicity of the triggering of the animated insertion because it avoided the use of a menu. S8 said “I don’t find the tool complex because there’s not really a menu, and that’s what makes tools complex sometimes. For the complexity, I would say it’s more about the fact that you have to press certain things, but then it’s just a mini entry ticket to know how to do this, so it’s not complex, I would say ”. As one of our participants commented, the interactions involving line insertions might look unusual initially, but it seems easy to get into the habit.

Since code is rearranged, it is worth noting that users may be disorientated. This was mentioned by one participant: “I clicked unintentionally, and I had trouble finding my way around, I didn’t know what I had clicked on” (S9). This is especially true in interactions with FSMs, as the layout may differ from the initial arrangement. We used animations to mitigate this aspect. We also provided an "original view" button to get the layout back to its original form. As for the FSM, the real-time animation was appreciated, but the reordered FSM animation was also found useful. S1, who liked the graphical FSM, underlined that the reordered FSM was very interesting. He said the drawback of the graphical FSM would be reproducing the error, but in graphic form. In his own words: “pre-mixed code is dangerous. But if you build it yourself, it forces you to reunderstand, it invites you to do it well, systematically. If it’s going to be a graphic tool, I think it should be something to build step by step, from a state, to build little by little, to check mentally.” That is why S1 found that the possibility to check state by state the in and out transitions for each state was a good compromise: “it is a tool, but it also forces you to check the correctness of the transitions”.

5.6.4.6 Design issues

The first type of issue concerns the GUI and UX design. Participants S1 and S4 criticized the use of colors as disturbing because one wonders what meaning they have. For now, the design choice associates a color randomly to each level of sources/destinations. S1 also found that it was unnecessary to have two bounce animations because it could confuse the eye. S1 and S4 suggest the tool would be better if there were a marker for the last source/final level, rather than having to test by clicking on the line and seeing whether it bounces. Some concerns about possible confusion and suggested solutions were made. The confusion was due more precisely to the fact that the developer could

get lost and forget where he originally clicked. Participants suggested it would then be helpful to change the color, enhance the first line clicked, or introduce some marker (S4). S7 insisted that the sources and destinations inserted should be somehow distinguished. S5 brought up another confusing issue about the real-time animation: the presence of too many bars simultaneously felt disturbing.

One main limit pointed out by S1, S3, S7, and S9 is that the starting point to figure out the causal chain should be in the GUI itself. For example, in the case of the bug related to the callsign display, participants mentioned they would have liked to click on the GUI where the callsign was supposed to be displayed and get information from there. Without such an option, a user might miss the link, which tells that the empty space also possesses a text property called "callsign2". As S7 puts it, "we want to go back and forth from the application itself to its code". Two participants referred to an existent tool, WPF Snoop, which allows the exploration of only the scene graph and graphical properties. The user clicks on the GUI and sees the corresponding tree, the values next to it, and the different graphical layers.

5.7 Threats to validity

Participants. The number of participants in the experiment is low, and they are all men. We could not avoid this sample bias, as the users of the Smala language happen to be men. Still, we do not have any plausible hypothesis about the effect of gender on the results of the experiments. As noted by Feitelson [91], the effect of gender is unclear and requires further investigation.

Size of code. The application used during the evaluation is 1,000 lines of code only. One can wonder if the interaction techniques would scale to larger code bases. 1,000 lines seem small, but they are written in a specialized, interaction-oriented programming language that is expressive and should not be compared to the size of code written in a general-purpose language such as Java. Still, the FSM of the example has a moderate complex behavior. Scaling up the number of states to a dozen would make it more difficult to perform interaction 3. Besides, our representations have limits: for example, Interaction 1 and 2 only display one causal chain at a time, and do not allow exploring multiple chains simultaneously. However, the control condition does not facilitate exploring one causal chain only, and we think it would scale even more badly, as one would have to navigate more files with longer causal chains.

Tasks. The tasks only partially represent the programming activity. We are aware that the evaluation has limits and should be complemented with other experiments. In particular, tasks A-B and C-D where the results were the most in favor of Causette, may seem idiosyncratic. However, the tasks represent a typical problem encountered by programmers of interactive programs. We designed the interactions to support solving those problems, hence it is not surprising that the evaluation suggests that Causette offers support. Nevertheless, it had to be demonstrated.

Control condition. The choice of a limited set of Sublime Text interactions as the control condition is disputable. Causette is a set of interaction techniques that may have been compared to a full-blown IDE. Still, the participants chose the Sublime Text interactions they use in their real-world activities. In particular, one can debug a program with Sublime Text. However, such a debugger is of little help for the interaction code problems. Comparing Causette with a full-featured IDE with a lot to cope with (window management, multiple routes, and options) would have been problematic. Therefore, we thought that a first step would be comparing Causette and the kind of restricted debugging tools available in Sublime (search, "go to definition" etc.), which is easier to use than a full-featured IDE. This still leaves several challenges to guarantee that comparing Causette and control conditions is fair. Causette involves run-time code visualization, but Sublime has no run-time debugging features.

Understanding. A general pitfall of comprehension evaluation [91] applies to our work: does the study say something about *comprehension*? It could turn out that some participants, especially for task A-B and C-D, found the *correct answers* about the causal chains in code, without properly *understanding* them. Addressing that issue would require further investigation.

Reading/Writing. Finally, the study suggests that Causette supports code reading, but not code writing. Since developers read 10 times more code than they write [91, 172], our results suggest that at least Causette could be useful. Further work is needed to assess whether Causette could support code writing.

Generalizability. The interactions may be too specialized to our particular language. Some languages or toolkits provide some of Smala's features, e.g., Qt's signal/slot, JavaFX binding, SwingState's FSMs. We think that most of the interactions could be applied to these features, e.g., navigating the chain of signal/slots in Qt or SwingState's FSMs. However, this necessitates appropriate analysis and introspection tools. For example, one could use Qt's MetaObject system to gather dependency information and use the API provided by some text editors to insert the upstream and downstream signal/slots where a particular connection is created in the code. Similarly, the QML or SwingStates run-times could record the parameters of the transitions and inform Causette. With this, it would be possible to adapt Causette interactions to an editor of Qt code: clicking on *object*1.*signal* in this line of code...:

```
object1.signal.connect(object2.slot)
```

...could summon the apparition of an upstream construct above:

```
object0.signal.connect(object1.slot)

object1.signal.connect(object2.slot)
```

Similarly, since QML and Swingstates provide explicit FSMs syntaxes, an editor of such languages could rearrange the order of transitions as in interaction 2.

5.8 Summary

Causette is a set of four interaction techniques for a textual and graphical code editor. The interactions rearrange and animate textual code to make the causal relationships more understandable. An evaluation with professional programmers suggests that Causette may be more usable than a regular text editor for some interaction programming tasks. This work indicates that rearranging interaction code may help developers better understand and fix it.

Causette matches practically the theoretical importance of the *causal orchestrator* defined in Chapter 4, as well as the insights gained on practice in Chapter 1. The causal orchestrator registers, guarantees and implements the causal relationships specified by the programmers. Causette, in that respect, allows the programmer to inspect the causal orchestrator's content, and check whether the described relationships matches without error his or her intentions.

This work can be continued by exploring unanswered questions on scalability, generalization, disorientation, and ecological validity. Instead of running another controlled experiment, a next step would consists in providing programmers with a better, more robust, and more integrated version of our tool and observe its use in practice during a longitudinal study. This should lead to more ecological results, but also to new insights on how to best support interaction programmers.

Conclusion

The motivation of this work stems from an observed mismatch between the classical conceptual framework in the epistemology of computing and practices when it comes to describing interactive computing from a software and hardware point of view. To address what interaction is, there are known socio-technical theories available within the HCI community, but less work targeting a theoretical counterpart of computability theory. Concepts of interaction expressiveness or completeness [44, 49], or abstract concepts and models dedicated to interactive systems [19, 77, 126, 212, 213] are not integrated within a general theory of interaction. If we look at theoretical computer science, the concept of interaction is emptied and broken down into debates on the reducibility of interactive systems to classical models of computation (by excellence, the TM). Some essential conceptual bricks are proposed in the literature on interactive system engineering. However, the proposed concepts are not posited at the same general level of abstraction as the TM is for computation.

Contributions

Our work is addressed to computer scientists in HCI and philosophers of computing. The former ask the question of what interaction is [22, 120, 153] or comes to be [19]. We propose as an answer — on the software and hardware sides — to conceptualize requirements for an execution model and identify its minimal components. Such abstractions allow articulating the relationships between interaction code and its execution at the same level of generality as the TM for computation. It allows, at the same time, to reflect on the expressiveness of interaction languages instead of computation-oriented ones. To the latter who wonder about the definition of a computer [39, 225, 265], and the bridging from abstraction to implementation [55, 187, 242, 254], we propose interaction as a case study that, we hope, brings at least some suggestions and arguments to discuss:

- A computing system must be understood as an interactive system. Understanding,

e.g., how a drawing application works, is the new fundamental *explanandum*[9], replacing the need to account for algorithmic and transformational systems.

- With only formal mathematical models, we are not provided with the right *explanans*[10], because we will not be able to explain essential aspects of the phenomenon for a causal description.
- An intermediate abstraction, like what we call an execution model, could be a candidate. It allows an intermediate level of description between the implementation and the instructions that the interaction programmer needs.
- This type of abstraction, as we suggest, takes into account practice (deduced from interviews with practitioners) and can support reflection on practice, e.g., a typology of current interaction languages and frameworks used.
- The interactive execution model suggests bricks to rethink a third way in the accounts of concrete computation [11]. It puts programmers at the forefront and addresses concrete computation by having a mechanistic account linking interactive instructions to their execution by a computing system.

Starting with a conceptual problem about computing, we approached it by combining HCI methods to investigate the reality of interaction programming practices (Chapter 1), with a philosophical analysis focused on models of interactive computation (Chapters 2 and 3). This combined approach helped flesh out the historical and conceptual reasons for the gap between interaction programming and the classical framework. It also demonstrated that there is no available abstraction doing for interactive systems what the TM does for transformational systems. We remind [12] that the singularity of the TM is to provide a formalism and an execution model, accounting for how computations described within a specific language are carried out. Based on Chapter 1 providing an overview of the interaction programming practices, landscape, issues, and concepts, we identify building blocks that support an execution model dedicated to interaction. The key aspect is the causal orchestrator, putting at the front stage the notion of *programmable behavior* (Chapter 4). It is the possibility of programming (timed) causal relationships between processes that makes computing interactive. The importance of causal understanding motivated the set of interaction techniques we presented in Chapter 5. The results foster causal understanding as a promising general guideline to support the comprehension and debugging of causal chains, in causal constructs such as dataflows and states machines.

We can sum up our contributions as follows:

[9] *Explanandum* = what needs to be explained.

[10] *Explanans* = the thing that *explains*, as opposed to the *explanandum*

[11] Concrete computation refers to computation carried out by physical systems, hence the adjective. For an introduction to concrete computation, see Piccinini's article in the Stanford Encyclopedia: `https://plato.stanford.edu/entries/computation-physicalsystems/`

[12] As previously said, comments on the twofold nature of the TM are well-known in philosophy and date back to Gödel [259].

- We have investigated specific challenges arising in interaction programming (through a study with professional programmers). We conceptualized the results by contrasting the stated issues with computation-oriented problems, suggesting interaction programmers were not only faced with algorithmic problems but were also significantly dealing with the physicality of the interactive computing system. We propose to label these latter issues as "causality" issues. Introducing the concept of "causality issues" allows for embracing many coding and understanding problems. It is reminiscent of a general requirement that has been stated in the literature [30, 207, 235]. It also serves as a starting point to think of the TM counterpart for interaction.

- We presented a novel state of the art, reviewing explicit models of interactive computing in theoretical computer science.

- Through an epistemological lens, we questioned whether we have a mechanistic explanation for interactive computing. We surveyed how interactive properties have been formalized and, with references to the philosophy of computing, conceptualized why these formalisms do not provide a satisfactory explanation.

- We proposed minimal components for an execution model. It helped us reflect on existing interaction languages. These abstractions can provide philosophers with a tool to describe, more specifically, how the execution of an interaction program comes about.

- We developed a tool to support causal understanding in interaction code through a set of interaction techniques: *Causette*. The tool is dedicated to the understanding and exploration of dataflow and FSMs. More precisely, our contributions are:

  - The elicitation of four design principles (5.3) we relied upon to design four interactions (5.4) that rearrange interaction code representations to support some interaction programming tasks;
  - The demonstration of the interactions in relevant scenarios (5.4);
  - The evaluation of the interactions with a semi-controlled quantitative and qualitative experiment with 10 professional programmers (5.6).

Implications

Some general implications can be drawn. First, our thesis supports the idea that interactive computing systems cannot be understood within a mere formal frame related to computability theory and its extensions. Second, it also fosters work on *programmability* in the philosophy of computing [294], a topic which is only starting to get attention [13].

[13] The topic has been brought up within the *History and Philosophy of Programming* (HaPoP) community. See `http://hapoc.org/publications`

Third, the results of the evaluation study in Chapter 5 encourage approaches adopting the concept of *causality understanding* to think of interaction programming. It helps understand and debug the causal chains that structure interactive behaviors. Finally, beyond the philosophy of computing, our view on interaction has possible implications for computational modeling. It makes an argument in favor of the distinction between *computational explanation* and *computational modeling* (as can be found in cognitive science [134, 189]): a mere formal computational model cannot explain (causally) a computational system. In that sense, interactive computing can serve as a case study to reflect on this useful distinction across computational disciplines.

Limitations

The first limitation is related to our interviews. Our interviewees were specialized in UI programming (except I8). They should be completed with more diverse uses of languages and frameworks. We did not cover all the varieties of the interaction programming landscape: for example, no interviewee had a significant experience in reactive functional or synchronous reactive programming. This should also be completed by exploring other forms of interaction programming not involving human users (e.g., server programming coded with open source server environment like Node.js).

We have motivated the need for a dedicated execution model from an epistemological point of view: understanding interaction. We have suggested that it also provides promising guidelines when developing tools for interaction programming. However, this conceptual motivation could be strengthened if we could further prove its practical interest. It would have to be proven (e.g., through studies and interviews) that the interaction execution model and the interaction expressiveness criterion actually ease the programmers in their practice. Would it help in the processing of learning interactive programming? Would it have an impact on formal work on interaction-oriented languages and frameworks?

Some limitations are then to be found in the conceptual rigor of the proposed "building blocks" for interaction. We have presented the necessary components of the execution model. Although we have arguments in their favor, drawn from interviews and a survey of practice and models in interactive system engineering, there is no formal proof of their necessity. Their sufficiency is yet to be proven. The status of the clock in particular is questionable, as it could be considered as a kind of transducer. Moreover, we should go further to reflect on the relation between our epistemological notion of execution model, with the concepts of abstract machine and execution model in operational semantics. Can we go further from the proposed reflection to formalize an interactive abstract machine at the defined level of abstraction (equivalent to the TM for classical computation)?

Finally, we have used an implicit and loose account of causality in the way it is commonly used in the HCI community [120, 136]. With the help of philosophy of science[294], we could have gone further and posit our concept of "causality engine" and "causal understanding" more precisely among theories of causation.

Future work

Future work would target the refinement of the execution model. Formal work would be warranted, e.g., comparing formalisms able to express the execution model. We think recent work on the formalization of dynamic causal relationships could be particularly relevant [12]. Another aspect would be to better specify the measurement of interaction expressiveness: how to value and weigh each aspect.

Future work would also enrich our thesis's philosophical content with current ongoing work in philosophy on programmability [196] and the search for an adequate causal account [294]. As already stated, that field is novel, but there are surely useful insights to be gained from the community.

Regarding Causette, we mentioned the limits of the proposed interaction techniques (Chapter 5, Threats to validity). Future work would not only involve overcoming these limits but also integrating new relevant interaction techniques to cover more of the interviewees' suggestions, e.g., easing the adjustment of frequency rates (as suggested by I5, I8, I10).

Broader scope and motivation

The scope of the thesis concerns the epistemology of computing through the lens of interaction programming practices. From a further distance, because we are concerned with the scope of the Turing Machine regarding interactive computing, our work carries some material to reflect on the use of the Turing Machine in computational modeling and explanation. References to the Turing Machine are pervasive beyond computing practices and influence the working hypotheses, e.g., in the philosophy of cognitive science. Whether the mind is a Turing machine or whether Turing himself saw his machine as possibly mimicking human intelligence is still addressed, especially in the AI community. Debates about the explanatory power of classical models of computation are also at stake to discuss the relevance of computational theories of mind. We think our work on interactive computing systems, although not aimed at modeling and explaining the mind, has interesting commonalities with these debates.

There have also been historical work to clarify Turing's own stance [222], arguing that Turing actually made no serious comparison between computing machines and minds, or that the Church Turing thesis entails nothing for computationalism [223]. However, the comparison between Turing machines and minds is still discussed: "On Minds and Turing Machines" [261], "Is the Mind A Turing Machine and How Could We Tell?" [188], "Is everything a Turing machine, and does it matter to the philosophy of mind?" [224]. Cognitive science has been using the concept of "computation" with strong connections to its birth in theoretical computer science. And although some might nuance the historical inheritance of the computation concept from computer science [6], it is hard to deny the influence of the Turing Machine in framing the notion of "computation" as used

in cognitive science. The reason is likely that among formal accounts of computation in computability theory, the TM intuitively relates to the cognitive activity, as it describes a human following a procedure to carry out some calculation [95]. Another reason for the influence of the TM is that it can also serve as a mechanical description for a *concrete computation*. Concrete computation is computation as actually implemented in a physical system. And the core challenge for philosophers of cognitive science is to provide an account of the mind bridging between the description of an abstract computation (mathematical) and (physical) concrete computation [95]. That is also why the TM in cognitive science, similarly to what we described in Chapter 3, has brought so much attention among other available formalisms. The TM also provides intuition about the execution of computation. In that respect, our work on formal models of computation and the limits we have pointed out share a lot with similar debates in cognitive science, where researchers like Fresco and Shagrir also remind us of the difference between (most) classical formal models of computation and concrete accounts [95].

The way our work could bring something to these discussions goes as follows: whether a computer is a Turing machine is a way less settled debate and probably a worth exploring one since "Cognitive science is based by and large on the premise that minds and computers have a great deal in common" [95]. Reflecting on what current computing systems do can have epistemological consequences in the philosophy of cognitive science: if computing machines themselves can be proven not to be Turing Machines in an explanatory sense, why would the mind be? By extension, our thoughts on interactive systems may help reflect the relationships between minds and computers, a question that is still at stake: "Why Think That The Brain Is Not A Computer?" [191] or "Why We View the Brain as a Computer" [258]. At least, if the analogy between minds and computers is to stay as a working hypothesis, more accounts of what computers are and do cannot be superfluous.

Bibliography

[1] Samson Abramsky. Information, processes and games. *Philosophy of Information*, 2008. doi: 10.1016/B978-0-444-51726-5.50017-0.

[2] Samson Abramsky. Robin milner's work on concurrency. In *Electronic Notes in Theoretical Computer Science*, 2010. doi: 10.1016/j.entcs.2010.08.002.

[3] Johnny Accot, Stéphane Chatty, Sébastien Maury, and Philippe Palanque. Formal transducers: Models of devices and building bricks for the design of highly interactive systems. In *Design, Specification and Verification of Interactive Systems' 97*, pages 143–159. Springer, 1997. doi: 10.1007/978-3-7091-6878-3_10.

[4] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1985. doi: 10.1007/3-540-16042-6_2.

[5] Gul A. Agha and Wooyoung Kim. Actors: A unifying model for parallel and distributed computing. In *Journal of Systems Architecture*, 1999. doi: 10.1016/S1383-7621(98)00067-8.

[6] Kenneth Aizawa. Computation in cognitive science: It is not all about turing-equivalent computation. *Studies in History and Philosophy of Science Part A*, 2010. doi: 10.1016/j.shpsa.2010.07.013.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994. doi: 10.1016/0304-3975(94)90010-8.

[8] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 1993. doi: 10.1006/inco.1993.1025.

[9] Henrik Reif Andersen, Simon Mørk, and Morten Ulrik Sørensen. A universal reactive machine. In *Lecture Notes in Computer Science (including subseries Lecture Notes*

in Artificial Intelligence and Lecture Notes in Bioinformatics), 1997. doi: 10.1007/3-540-63141-0_7.

[10] Caroline Appert and Michel Beaudouin-Lafon. Swingstates: Adding state machines to the swing toolkit. In *UIST 2006: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, 2008. doi: 10.1145/1166253.1166302.

[11] Caroline Appert, Stéphane Huot, Pierre Dragicevic, and Michel Beaudouin-Lafon. Flowstates: Prototypage d'applications interactives avec des flots de données et des machines à États. In *Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine*, pages 119–128. Association for Computing Machinery, 2009. doi: 10.1145/1629826.1629845.

[12] Youssef Arbach, David Karcher, Kirstin Peters, and Uwe Nestmann. Dynamic causality in event structures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015. doi: 10.1007/978-3-319-19195-9_6.

[13] John Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21(8), 1978. doi: 10.1145/359576.359579.

[14] Jos C.M. Baeten, Bas Luttik, and Paul Van Tilburg. Turing meets milner. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012. doi: 10.1007/978-3-642-32940-1_1.

[15] Jos C.M. Baeten, Bas Luttik, and Paul Van Tilburg. Reactive turing machines. *Information and Computation*, 2013. doi: 10.1016/j.ic.2013.08.010.

[16] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2013. doi: 10.1145/2501654.2501666.

[17] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I.* 1988. doi: 10.1007/978-3-642-97062-7.

[18] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual petri nets, asymmetric event structures, and processes. *Information and Computation*, 2001. doi: 10.1006/inco.2001.3060.

[19] Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. An anatomy of interaction: Co-occurrences and entanglements. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 188–196. Association for Computing Machinery, 2018. doi: 10.1145/3191697.3214328.

[20] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *Proceedings of the Workshop on Advanced Visual Interfaces AVI*, 2004. ISBN 1581138679. doi: 10.1145/989863.989865.

[21] Michel Beaudouin-Lafon. Human-computer interaction. In *Interactive Computation: The New Paradigm*, 2006. doi: 10.1007/3-540-34874-3_10.

[22] Michel Beaudouin-Lafon. Interaction is the future of computing. In T. Erickson and D. McDonald, editors, *HCI Remixed, Reflections on Works That Have Influenced the HCI Community*, pages 263–266, 2008.

[23] Michel Beaudouin-Lafon, Susanne Bødker, and Wendy E Mackay. Generative theories of interaction. *ACM Trans. Comput.-Hum. Interact.*, 28, 11 2021. doi: 10.1145/3468505.

[24] William Bechtel. Levels of description and explanation in cognitive science. *Minds and Machines*, 1994. doi: 10.1007/BF00974201.

[25] A Benveniste, P Caspi, S A Edwards, N Halbwachs, P Le Guernic, and R de Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, 1 2003. doi: 10.1109/JPROC.2002.805826.

[26] Gérard Berry. Real time programming: Special purpose or general purpose languages, 1989.

[27] Gérard Berry. *The Constructive Semantics of Pure Esterel*. 2002. URL `http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf`.

[28] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 1992. doi: 10.1016/0167-6423(92)90005-V.

[29] Gérard Berry and Manuel Serrano. Hop and hiphop: Multitier web orchestration. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. doi: 10.1007/978-3-319-04483-5_1.

[30] Gérard Berry and Manuel Serrano. Hiphop.js: (a)synchronous reactive web programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020. doi: 10.1145/3385412.3385984.

[31] Jacques Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983. ISBN 0299090604.

[32] L. Thomas Van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs Van Der Storm, Benoit Combemale, and Olivier Barais. A principled approach to repl interpreters. In *Onward! 2020 - Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-located with SPLASH 2020*, 2020. doi: 10.1145/3426428.3426917.

[33] Alan Blackwell and Thomas Green. Notational systems-the cognitive dimensions of notations framework. In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, 2003. doi: 10.1016/B978-155860808-5/50005-8.

[34] Renaud Blanch and Michel Beaudouin-Lafon. Programming rich interactions using the hierarchical state machine toolkit. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, page 51–58, New York, NY, USA, 2006. Association for Computing Machinery. doi: 10.1145/1133265.1133275.

[35] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto. *Reactivemanifesto.Org*, 2014.

[36] Worth Boone and Gualtiero Piccinini. Mechanistic abstraction. *Philosophy of Science*, 2016. doi: 10.1086/687855.

[37] Frédéric Boussinot. Reactive c: An extension of c to program reactive systems. *Software: Practice and Experience*, 21:401–428, 1991. doi: 10.1002/spe.4380210406.

[38] Robert Boyer and Jstrother Moore. A mechanical proof of the turing completeness of pure lisp. *Automated Theorem Proving: After 25 Years*, 8 1997. doi: 10.1090/conm/029/08.

[39] Cem Bozşahin. Computers aren't syntax all the way down or content all the way up. *Minds and Machines*, 2018. doi: 10.1007/s11023-018-9469-2.

[40] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings - International Conference on Software Engineering*, 2010. doi: 10.1145/1806799.1806866.

[41] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. Laviola. Code bubbles: A working set-based interface for code understanding and maintenance. In *Conference on Human Factors in Computing Systems - Proceedings*, 2010. doi: 10.1145/1753326.1753706.

[42] Alexander Breckel and Matthias Tichy. Embedding programming context into source code. In *IEEE International Conference on Program Comprehension*, 2016. doi: 10.1109/ICPC.2016.7503732.

[43] John Brooke. Sus: A 'quick and dirty' usability scale. In Weerdmeester B.A. McClelland A.L. Jordan P.W., Thomas B, editor, *Usability Evaluation In Industry*. Taylor and Francis, 1986.

[44] Philippe Brun and Michel Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of interactive systems. In *Proceedings of the HCI'95 Conference on People and Computers X*, page 197–212. Cambridge University Press, 1996.

[45] Maarten Bullynck, Edgar G Daylight, and Liesbeth De Mol. Why did computer science make a hero out of turing? *Commun. ACM*, 58:37–39, 2 2015. doi: 10.1145/2658985.

[46] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *IEEE International Conference on Program Comprehension*, 2015. ISBN 9781467381598. doi: 10.1109/ICPC.2015.36.

[47] Tim Button and Sean Walsh. *Philosophy and Model Theory*. 2018. ISBN 9780198790396. doi: 10.1093/oso/9780198790396.001.0001.

[48] Pascal Béger. *Vérification formelle des propriétés graphiques des systèmes informatiques interactifs*. PhD thesis, 2020.

[49] José C Campos and Michael D Harrison. Formally verifying interactive systems: A review. *Design, Specification and Verification of Interactive Systems' 97*, pages 109–124, 1997.

[50] Alexandre Canny, David Navarre, José Creissac Campos, and Philippe Palanque. Model-based testing of post-wimp interactions using object oriented petri-nets. In *International Symposium on Formal Methods*, pages 486–502, 2019. doi: 10.1007/978-3-030-54994-7_35.

[51] Géry Casiez and Nicolas Roussel. No more bricolage! methods and tools to characterize, replicate and compare pointing transfer functions. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 603–614. Association for Computing Machinery, 2011. doi: 10.1145/2047196.2047276.

[52] Géry Casiez, Stéphane Conversy, Matthieu Falce, Stéphane Huot, and Nicolas Roussel. Looking through the eye of the mouse: A simple method for measuring end-to-end latency using an optical mouse. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software amp; Technology*, pages 629–636. Association for Computing Machinery, 2015. doi: 10.1145/2807442.2807454.

[53] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New

York, NY, USA, 1987. Association for Computing Machinery. doi: 10.1145/41625.41641.

[54] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming with lucid synchrone. In *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, 2010. doi: 10.1002/9780470611012.ch7.

[55] David J. Chalmers. The varieties of computation: A reply. *Journal of Cognitive Science*, 2012. doi: 10.17791/jcs.2012.13.3.211.

[56] Stéphane Chatty. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, pages 195–204. Association for Computing Machinery, 1994. doi: 10.1145/192426.192500.

[57] Stéphane Chatty. Programs = data + algorithms + architecture: Consequences for interactive software engineering. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008. doi: 10.1007/978-3-540-92698-6_22.

[58] Stéphane Chatty and Stéphane Conversy. What programming languages for interactive systems designers? In *Engineering Interactive Computing Systems Workshop (EICS)*, 2014. URL `https://hal-enac.archives-ouvertes.fr/hal-01024013`.

[59] Elizabeth Chell. Critical incident technique. In G. Symon and C. Cassell, editors, *Qualitative methods and analysis in organizational research: A practical guide*, pages 51–72. Sage Publications Ltd., 1998.

[60] Fanny Chevalier, Pierre Dragicevic, Anastasia Bezerianos, and Jean Daniel Fekete. Using text animated transitions to support navigation in document histories. In *Conference on Human Factors in Computing Systems - Proceedings*, 2010. doi: 10.1145/1753326.1753427.

[61] Fanny Chevalier, Nathalie Henry Riche, Catherine Plaisant, Amira Chalbi, and Christophe Hurter. Animations 25 years later: New roles and opportunities. In Paolo Buono, Rosa Lanzilotti, Maristella Matera, and Maria Francesca Costabile, editors, *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2016, Bari, Italy, June 7-10, 2016*, pages 280–287. ACM, 2016. doi: 10.1145/2909132.2909255.

[62] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 1936. doi: 10.2307/2371045.

[63] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 1940. doi: 10.2307/2266170.

[64] Paul Cockshott and Greg Michaelson. Are there new models of computation? reply to wegner and eberbach. *Computer Journal*, 2007. doi: 10.1093/comjnl/bxl062.

[65] Jean Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software, EMSOFT 2005*, 2005. doi: 10.1145/1086228.1086261.

[66] Stéphane Conversy. Contributions to the science of controlled transformations. hdr report, 2013.

[67] Stéphane Conversy. Unifying textual and visual: A theoretical account of the visual perception of programming languages. In *Onward! 2014 - Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2014*, 2014. doi: 10.1145/2661136.2661138.

[68] Stéphane Conversy, Géry Casiez, Matthieu Falce, Stéphane Huot, and Nicolas Roussel. US Patent App. 15/769,389.

[69] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, pages 294–308. Springer-Verlag, 2006. doi: 10.1007/11693024_20.

[70] B. Jack Copeland. Hypercomputation. *Minds and Machines*, 2002. doi: 10.1023/A:1021105915386.

[71] B. Jack Copeland and Oron Shagrir. Do accelerating turing machines compute the uncomputable? *Minds and Machines*, 2011. doi: 10.1007/s11023-011-9238-y.

[72] B. Jack Copeland and Oron Shagrir. The church-turing thesis. *Communications of the ACM*, 2018. doi: 10.1145/3198448.

[73] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013. doi: 10.1145/2462156.2462161.

[74] Martin Davis. Mathematical logic and the origin of modern computers. 1988. doi: 10.1007/978-3-7091-6597-3_5.

[75] Edgar G. Daylight. A turing tale. *Communications of the ACM*, 2014. doi: 10.1145/2629499.

[76] Edgar G. Daylight. Towards a historical notion of 'turing—the father of computer science'. *History and Philosophy of Logic*, 2015. doi: 10.1080/01445340.2015.1082050.

[77] Andrew M. Dearden and Michael D. Harrison. Abstract models for hci. *International Journal of Human Computer Studies*, 1997. doi: 10.1006/ijhc.1996.0087.

[78] Michael Desmond, Margaret Anne Storey, and Chris Exton. Fluid source code views. In *IEEE International Conference on Program Comprehension*, 2006. doi: 10.1109/ICPC.2006.24.

[79] L. Peter Deutsch and Edmund Berkeley. The lisp implementation for the pdp-1 computer, 1964. URL `https://www.computerhistory.org/pdp-1/1822b607c479d2e9de9b19ba958c16e3/`.

[80] Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. Evaluating a visual approach for understanding javascript source code. In *IEEE International Conference on Program Comprehension*, 2020. doi: 10.1145/3387904.3389275.

[81] E W Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569–, 9 1965. doi: 10.1145/365559.365617.

[82] Gordana Dodig-Crnkovic. Significance of models of computation, from turing model to natural computation. *Minds and Machines*, 2011. doi: 10.1007/s11023-011-9235-1.

[83] Pierre Dragicevic. Fair statistical communication in hci. In *Modern Statistical Methods for HCI*, pages 291 – 330. Springer, 2016. doi: 10.1007/978-3-319-26633-6_13.

[84] Pierre Dragicevic and Jean Daniel Fekete. Support for input adaptability in the icon toolkit. In *ICMI'04 - Sixth International Conference on Multimodal Interfaces*, 2004. doi: 10.1145/1027933.1027969.

[85] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. Gliimpse: Animating from markup code to rendered documents and vice versa. In *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 2011. doi: 10.1145/2047196.2047229.

[86] Carl Eastlund and Matthias Felleisen. Automatic verification for interactive graphical programs. In *ACM International Conference Proceeding Series*, 2009. ISBN 9781605587424. doi: 10.1145/1637837.1637843.

[87] Eugene Eberbach, Dina Goldin, and Peter Wegner. Turing's ideas and models of computation. In Christof Teuscher, editor, *Alan Turing: Life and Legacy of a Great Thinker*, pages 159–194. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-662-05642-4_7.

[88] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 1997. doi: 10.1145/258949.258973.

[89] Niklas Elmqvist and Philippas Tsigas. Causality visualization using animated growing polygons. In *Proceedings - IEEE Symposium on Information Visualization, INFO VIS*, 2003. doi: 10.1109/INFVIS.2003.1249025.

[90] Niklas Elmqvist, Andrew Vande Moere, Hans Christian Jetter, Daniel Cernea, Harald Reiterer, and T. J. Jankun-Kelly. Fluid interaction for information visualization. *Information Visualization*, 2011. ISSN 14738716. doi: 10.1177/1473871611413180.

[91] Dror G. Feitelson. Considerations and pitfalls in controlled experiments on code comprehension. In *IEEE International Conference on Program Comprehension*, 2021. doi: 10.1109/ICPC52881.2021.00019.

[92] Jean Daniel Fekete, Martin Richard, and Pierre Dragicevic. Specification and verification of interactors: A tour of esterel. In *Formal Aspects of Human Computer Interaction Workshop*, pages 25–32. Sheffield University, 1998.

[93] J.D Foley, A. van Dam, and S.K. Feiner. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[94] Nir Fresco. Explaining computation without semantics: Keeping it simple. *Minds and Machines*, 20:165–181, 2010. doi: 10.1007/s11023-010-9199-6.

[95] Nir Fresco. Concrete digital computation: What does it take for a physical system to compute? *Journal of Logic, Language and Information*, 20:513, 2011. doi: 10.1007/s10849-011-9147-8.

[96] Nir Fresco. A critical survey of some competing accounts of concrete digital computation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013. doi: 10.1007/978-3-642-44958-1-12.

[97] Maurizio Gabbrielli and Simone Martini. Abstract machine. In *Programming Languages: Principles and Paradigms*. Springer-Verlag, 2010. ISBN 978-1-84882-914-5.

[98] Rob Van Glabbeek and Gordon Plotkin. Event structures for resolvable conflict. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2004. doi: 10.1007/978-3-540-28629-5_42.

[99] Stuart Glennan. Rethinking mechanistic explanation. *Philosophy of Science*, 2002. doi: 10.1086/341857.

[100] M. D. Godfrey and D. F. Hendry. The computer as von neumann planned it. *IEEE Annals of the History of Computing*, 1993. doi: 10.1109/85.194088.

[101] Pascal Goffin, Wesley Willett, Jean Daniel Fekete, and Petra Isenberg. Exploring the placement and design of word-scale visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2014. doi: 10.1109/TVCG.2014.2346435.

[102] Dina Goldin and Peter Wegner. The interactive nature of computing: Refuting the strong church-turing thesis. *Minds and Machines*, 2008. doi: 10.1007/s11023-007-9083-1.

[103] Dina Goldin, Peter Wegner, and Scott A. Smolka. *Interactive Computation: The New Paradigm*. 2006. doi: 10.1007/3-540-34874-3.

[104] Dina Q. Goldin. Persistent turing machines as a model of interactive computation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2000. doi: 10.1007/3-540-46564-2_8.

[105] Dina Q. Goldin, Scott A. Smolka, and Peter Wegner. Turing machines, transition systems, and interaction. In *Electronic Notes in Theoretical Computer Science*, 2002. doi: 10.1016/S1571-0661(04)00220-8.

[106] R. L. Goodstein and Hartley Rogers. Theory of recursive functions and effective computability. *The Mathematical Gazette*, 1969. doi: 10.2307/3614588.

[107] Paul Graham. *Hackers and Painters: Essays on the Art of Programming*. O'Reilly Associates, Inc., USA, 2004.

[108] T.R.G. Green and M. Petre. When visual programs are harder to read than textual programs. *Human-Computer Interaction: Tasks and Organisation*, 1992.

[109] Valentina Grigoreanu, Roland Fernandez, Kori Inkpen, and George Robertson. What designers want: Needs of interactive application designers. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 139–146, 2009. doi: 10.1109/VLHCC.2009.5295277.

[110] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 1991. doi: 10.1109/5.97301.

[111] Thomas Haigh. 'stored program concept' considered harmful: History and historiography. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013. doi: 10.1007/978-3-642-39053-1_28.

[112] Thomas Haigh. Actually, turing did not invent the computer. *Communications of the ACM*, 2014. doi: 10.1145/2542504.

[113] Thomas Haigh and Mark Priestley. Historical reflections von neumann thought turing's universal machine was 'simple and neat.' but that didn't tell him how to design a computer. *Communications of the ACM*, 2020. doi: 10.1145/3372920.

[114] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, 1985. doi: 10.1007/978-3-642-82453-1_17.

[115] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. doi: 10.1016/0167-6423(87)90035-9.

[116] Juris Hartmanis. Turing award lecture on computational complexity and the nature of computer science. *Communications of the ACM*, 1994. doi: 10.1145/194313.214781.

[117] Ralph D Hill. Supporting concurrency, communication, and synchronization in human-computer interaction—the sassafras uims. *ACM Trans. Graph.*, 5:179–210, 7 1986. doi: 10.1145/24054.24055.

[118] Andrew Hodges. Did church and turing have a thesis about machines? In *Church's Thesis After 70 Years*, 2013. doi: 10.1515/9783110325461.242.

[119] Karen Holtzblatt, Jessamyn Wendell, and Shelley Wood. *Rapid Contextual Design*. 2005. doi: 10.1016/B978-0-12-354051-5.X5000-9.

[120] Kasper Hornbaek and Antti Oulasvirta. What is interaction? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 5040–5052. Association for Computing Machinery, 2017. doi: 10.1145/3025453.3025765.

[121] Scott E. Hudson, Roy Rodenstein, and Ian Smith. Debugging lenses: A new class of transparent tools for user interface debugging. In *UIST (User Interface Software and Technology): Proceedings of the ACM Symposium*, 1997.

[122] Stéphane Huot, Cédric Dumas, Pierre Dragicevic, Jean-Daniel Fekete, and Gérard Hégron. The magglite post-wimp toolkit: draw it, connect it and run it. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 257–266, 2004. doi: 10.1145/1029632.1029677.

[123] Daniel D. Hutto, Erik Myin, Anco Peeters, and Farid Zahnoun. The cognitive basis of computation: Putting computation in its place. *The Routledge Handbook of the Computational Mind*, 2018.

[124] ISO. Ergonomics of human-system interaction. part 11: Usability: Definitions and concepts. *ISO 9241-11:2018*, 2018.

[125] Robert J. K. Jacob. Human-computer interaction: Input devices. *ACM Comput. Surv.*, 28(1):177–179, mar 1996. doi: 10.1145/234313.234387.

[126] Robert J K Jacob, Leonidas Deligiannidis, and Stephen Morrison. A software model and specification language for non-wimp user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 6:1–46, 3 1999. doi: 10.1145/310641.310642.

[127] Ahmad Jbara. Simplyhover: Improving comprehension of else statements. In *IEEE International Conference on Program Comprehension*, 2020. doi: 10.1145/3387904.3389297.

[128] Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*, 2014. doi: 10.1145/2597008.2597140.

[129] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1:22–35, 1988.

[130] Neil D. Jones and Jakob Grue Simonsen. Programs = data = first-class citizens in a computational world. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 2012. doi: 10.1098/rsta.2011.0328.

[131] Peggy Aldrich Kidwell and William Aspray. John von neumann and the origins of modern computing. *Technology and Culture*, 1992. doi: 10.2307/3105832.

[132] Eugen Kiss. 7guis: A gui programming benchmark, 2014. URL `https://eugenkiss.github.io/7guis/`.

[133] Eugen Kiss. Towards a better gui programming benchmark, 2018. URL `https://hackernoon.com/towards-a-better-gui-programming-benchmark-397aca3542b8`.

[134] Colin Klein. Polychrony and the process view of computation. *Philosophy of Science*, 2020. ISSN 1539767X. doi: 10.1086/710613.

[135] Donald E. Knuth. *The Art of Computer Programming, Vol.1: Fundamental Algorithms*. 1968.

[136] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Conference on Human Factors in Computing Systems - Proceedings*, 2004. doi: 10.1145/985692.985712.

[137] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings - International Conference on Software Engineering*, 2008. doi: 10.1145/1368088.1368130.

[138] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 2006. doi: 10.1109/TSE.2006.116.

[139] Janusz Kowalik. Actors: A model of concurrent computation in distributed systems (gul agha). *SIAM Review*, 1988. doi: 10.1137/1030027.

[140] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Commun. ACM*, volume 21, pages 558–565. ACM, 7 1978. doi: 10.1145/359545.359563.

[141] Jean Lassègue and Giuseppe Longo. What is turing's comparison between mechanism and writing worth? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012. doi: 10.1007/978-3-642-30870-3_46.

[142] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings - International Conference on Software Engineering*, 2010. doi: 10.1145/1806799.1806829.

[143] Edward A Lee. Cyber-physical systems - are computing foundations adequate? *Position paper for NSF workshop on cyber-physical systems: research motivation, techniques and roadmap*, 2:1–9, 2006. URL `https://ptolemy.berkeley.edu/publications/papers/06/CPSPositionPaper/`.

[144] Edward A. Lee. Computing needs time. *Communications of the ACM*, 2009. doi: 10.1145/1506409.1506426.

[145] Edward A. Lee. Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, 2016. doi: 10.1145/2912149.

[146] Edward A. Lee. *The Coevolution*. MIT Press, 2020. doi: 10.7551/mitpress/12307.001.0001.

[147] Edward A. Lee. *Plato and The Nerd*. MIT Press, 2020. doi: 10.7551/mitpress/11180.001.0001.

[148] Edward A. Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. *System-on-Chip: Next Generation Electronics*, 2006. doi: 10.1049/PBCS018E_ch7.

[149] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1998. doi: 10.1109/43.736561.

[150] Jan Van Leeuwen and Jiří Wiedermann. On algorithms and interaction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2000. doi: 10.1007/3-540-44612-5_7.

[151] Jan Van Leeuwen and Jiří Wiedermann. Beyond the turing limit: Evolving interactive systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001. doi: 10.1007/3-540-45627-9_8.

[152] Jan Van Leeuwen and Jiří Wiedermann. A theory of interactive computation. In *Interactive Computation: The New Paradigm*, 2006. doi: 10.1007/3-540-34874-3_6.

[153] Catherine Letondal, Stéphane Chatty, W Greg Phillips, and Fabien André. Usability requirements for interaction-oriented development tools. In *PPIG*, 2010. URL `https://hal-enac.archives-ouvertes.fr/hal-01022441/file/253.pdf`.

[154] Nancy Leveson. Are you sure your software will not kill anyone? *Commun. ACM*, 63(2):25–28, jan 2020. doi: 10.1145/3376127.

[155] Tom Lieber. Theseus: Understanding asynchronous code. In *Conference on Human Factors in Computing Systems - Proceedings*, 2013. doi: 10.1145/2468356.2479501.

[156] Tom Lieber, Joel Brandt, and Robert C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Conference on Human Factors in Computing Systems - Proceedings*, 2014. doi: 10.1145/2556288.2557409.

[157] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Conference on Human Factors in Computing Systems - Proceedings*, 1995. doi: 10.1145/223904.223969.

[158] Giuseppe Longo. The difference between clocks and turing machines. In *Functional Models of Cognition*, 1999. doi: 10.1007/978-94-015-9620-6_14.

[159] Bas Luttik and Fei Yang. On the executability of interactive computation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016. doi: 10.1007/978-3-319-40189-8_32.

[160] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid i/o automata. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1996. ISBN 354061155X. doi: 10.1007/BFb0020971.

[161] Peter Machamer, Lindley Darden, and Carl F. Craver. Thinking about mechanisms. *Philosophy of Science*, 2000. doi: 10.1086/392759.

[162] Bruce MacLennan. Transcending turing computability. *Minds and Machines*, 2003. doi: 10.1023/A:1021397712328.

[163] Bruce MacLennan. Super-turing or non-turing? extending the concept of computation. *International Journal of Unconventional Computing*, 2009.

[164] Mathieu Magnaudet, Stephane Chatty, Stephane Conversy, Sebastien Leriche, Celia Picard, and Daniel Prun. Djnn/smala: A conceptual framework and a language for interaction-oriented programming. *Proceedings of the ACM on Human-Computer Interaction*, 2018. doi: 10.1145/3229094.

[165] Ingo Maier and Martin Odersky. Deprecating the observer pattern with scala.react, 2012.

[166] Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *European Conference on Object-Oriented Programming*, pages 707–731. Springer, 2013. doi: 10.1007/978-3-642-39038-8_29.

[167] Louis Mandel and Florence Plateau. Interactive programming of reactive systems. *Electronic Notes in Theoretical Computer Science*, 2009. doi: 10.1016/j.entcs.2008.01.004.

[168] Louis Mandel and Marc Pouzet. Reactiveml, un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques*, 27:1097–1128, 10 2008. doi: 10.3166/tsi.27.1097-1128.

[169] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. 1992. doi: 10.1007/978-1-4612-0931-7.

[170] Alessandro Margara and Guido Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. *Soft.Vub.Ac.Be*, 2013.

[171] David Marr. *Vision: A Computational Approach*. Freeman and co., 1982.

[172] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftmanship*. 2014.

[173] Simone Martini. The standard model for programming languages: The birth of a mathematical theory of computation. In *Recent Developments in the Design and Implementation of Programming Languages*, 2020. URL `https://hal.inria.fr/hal-03028634`.

[174] Anneliese Von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *Papers Presented at the 7th Workshop on Empirical Studies of Programmers, ESP 1997*, 1997. doi: 10.1145/266399.266414.

[175] J. Mccarthy. A basis for a mathematical theory of computation. *Studies in Logic and the Foundations of Mathematics*, 1959. doi: 10.1016/S0049-237X(09)70099-0.

[176] Ron McClamrock. Marr's three levels: A re-evaluation. *Minds and Machines*, 1: 185–196, 1990. doi: 10.1007/BF00361036.

[177] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 2009. doi: 10.1145/1640089.1640091.

[178] Microsoft. Codelens, 2013. URL `https://docs.microsoft.com/en-us/visualstudio/ide/find-code-changes-and-other-history-with-codelens?view=vs-2022`.

[179] Robin Milner. Processes: A mathematical model of computing agents. In H E Rose and J C Shepherdson, editors, *Logic Colloquium '73*, volume 80, pages 157–173. Elsevier, 1975. doi: 10.1016/S0049-237X(08)71948-7.

[180] Robin Milner. Four combinators for concurrency. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 1982. doi: 10.1145/800220.806687.

[181] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 1983. doi: 10.1016/0304-3975(83)90114-7.

[182] Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 1993. doi: 10.1145/151233.151240.

[183] Robin Milner. *Communicating and Mobile Systems: the -calculus*. Cambrdige University Press, 1999.

[184] Robin Milner. Turing, computing and communication. 2006. doi: 10.1007/3-540-34874-3_1.

[185] Robin Milner. *The Space and Motion of Communicating Agents*. 2009. ISBN 9780511626661. doi: 10.1017/CBO9780511626661.

[186] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. Taming the ide with fine-grained interaction data. In *IEEE International Conference on Program Comprehension*, 2016. doi: 10.1109/ICPC.2016.7503714.

[187] Marcin Miłkowski. Beyond formal structure: A mechanistic perspective on computation and implementation. *Journal of Cognitive Science*, 2011.

[188] Marcin Miłkowski. Is the mind a turing machine? how could we tell? In *Church's Thesis. Logic, Mind, and Nature*, 2014.

[189] Marcin Miłkowski. Computational mechanisms and models of computation. *Philosophia Scientiae*, 2014. doi: 10.4000/philosophiascientiae.1019.

[190] Marcin Miłkowski. A mechanistic account of computational explanation in cognitive science and computational neuroscience. In *Computing and Philosophy*, 2016. doi: 10.1007/978-3-319-23291-1_13.

[191] Marcin Miłkowski. Why think that the brain is not a computer? *APA Newsletter on Philosophy and Computers*, 16(2):22–28, 2016.

[192] Marcin Miłkowski. Objections to computationalism: A survey. 2018. doi: 10.18290/rf.2018.66.3-3.

[193] Liesbeth De Mol. Turing machines. *Stanford Encyclopedia*, 2018. URL `https://plato.stanford.edu/entries/turing-machine/`.

[194] Liesbeth De Mol and Maarten Bullynck. Making the history of computing. the history of computing in the history of technology and the history of mathematics. *Revue de Synthèse*, 139:361 – 380, 2018. doi: https://doi.org/10.1163/19552343-13900017.

[195] Liesbeth De Mol and Giuseppe Primiero. When logic meets engineering: Introduction to logical issues in the history and philosophy of computer science. *History and Philosophy of Logic*, 36:195–204, 2015.

[196] Liesbeth De Mol, Bullynck Maarten, and Edgar G. Daylight. Less is more in the fifties: Encounters between logical minimalism and computer design during the 1950s. *IEEE Annals of the History of Computing*, 2018. doi: 10.1109/MAHC.2018.012171265.

[197] Daniel Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. 2009. doi: 10.1109/TSE.2009.67.

[198] David L Morgan and Robin K Morgan. Single-case research methods for the behavioral and health sciences. In *Research Methods in Psychology*, 2017. URL `https://opentext.wsu.edu/carriecuttler/chapter/overview-of-single-subject-research/`.

[199] Pierre Mounier-Kuhn. Logic and computing in france: A late convergence. In *AISB/IACAP World Congress 2012: Symposium on the History and Philosophy of Programming, Part of Alan Turing Year 2012*, 2012. ISBN 9781908187178.

[200] Brad Myers. Challenges of hci design and implementation. *interactions*, 1994. ISSN 15583449. doi: 10.1145/174800.174808.

[201] Brad Myers, Scott E Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, 3 2000. doi: 10.1145/344949.344959.

[202] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, 2008. doi: 10.1109/VLHCC.2008.4639081.

[203] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, 2008. doi: 10.1109/VLHCC.2008.4639081.

[204] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991*, 1991.

[205] Brad A. Myers. Improving program comprehension by answering questions (keynote). In *IEEE International Conference on Program Comprehension*, 2013. doi: 10.1109/ICPC.2013.6613827.

[206] Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Vander Zanden, David Kosbie, Richard McDaniel, James Landay, Matthew Goldberg, and Rajan Pathasarathy. The garnet user interface development environment. In *Conference on Human Factors in Computing Systems - Proceedings*, 1994. doi: 10.1145/259963.260472.

[207] Brad A Myers, Dario A Giuse, Roger B Dannenberg, Brad Vander Zanden, David S Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet comprehensive support for graphical, highly interactive user interfaces. In Ronald Baecker, Jonathan Grudin, William Buxton, and Saul Greenberg, editors, *Readings in Human–Computer Interaction*, pages 357–371. Morgan Kaufmann, 1995. doi: 10.1016/B978-0-08-051574-8.50037-6.

[208] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, and Bruce D. Kyle. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 1997. doi: 10.1109/32.601073.

[209] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Communications of the ACM*, 2004. doi: 10.1145/1015864.1015888.

[210] Brad A. Myers, Andrew J. Ko, Thomas D. Latoza, and Youngseok Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 2016. ISSN 00189162. doi: 10.1109/MC.2016.200.

[211] E. N., D. Hilbert, and W. Ackermann. Grundzuge der theoretischen logik. *The Journal of Philosophy*, 1938. doi: 10.2307/2018808.

[212] David Navarre, Philippe Palanque, Pierre Dragicevic, and Rémi Bastide. An approach integrating two complementary model-based environments for the construction of multimodal interactive applications. *Interacting with Computers*, 18:910–941, 2006. doi: https://doi.org/10.1016/j.intcom.2006.03.002.

[213] David Navarre, Philippe Palanque, Jean Francois Ladry, and Eric Barboni. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction*, 2009. doi: 10.1145/1614390.1614393.

[214] Maxwell Herman Alexander Newman. Alan mathison turing, 1912-1954. *Biographical Memoirs of Fellows of the Royal Society*, 1955. doi: 10.1098/rsbm.1955.0019.

[215] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 1981. doi: 10.1016/0304-3975(81)90112-2.

[216] Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles (History of Computing S.)*. MIT Press, 2008. ISBN 9780262640688.

[217] DA Norman and Stephen W Draper. User centered system design - new perspectives on human-computer interaction edited by. *University of California, San Diego*, 1986.

[218] Stephen Oney, Brad Myers, and Joel Brandt. Interstate: A language and environment for expressing interface behavior. In *UIST 2014 - Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, 2014. ISBN 9781450330695. doi: 10.1145/2642918.2647358.

[219] Antti Oulasvirta, Per Ola Kristensson, Xiaojun Bi, and Andrew Howes. *Computational Interaction*. Oxford University Press. ISBN 9780198799603. doi: 10.1093/oso/9780198799603.001.0001.

[220] Norman Peitek, Janet Siegmund, and Sven Apel. What drives the reading order of programmers? an eye tracking study. In *IEEE International Conference on Program Comprehension*, 2020. doi: 10.1145/3387904.3389279.

[221] C. A. Petri. Introduction to general net theory. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1980. doi: 10.1007/3-540-10001-6_21.

[222] Gualtiero Piccinini. Alan turing and the mathematical objection. *Minds and Machines*, 2003. doi: 10.1023/A:1021348629167.

[223] Gualtiero Piccinini. Computing mechanisms. *Philosophy of Science*, 2007. doi: 10.1086/522851.

[224] Gualtiero Piccinini. Computational modelling vs. computational explanation: Is everything a turing machine, and does it matter to the philosophy of mind? *Australasian Journal of Philosophy*, 2007. doi: 10.1080/00048400601176494.

[225] Gualtiero Piccinini. Computers. *Pacific Philosophical Quarterly*, 2008. doi: 10.1111/j.1468-0114.2008.00309.x.

[226] Gualtiero Piccinini. Some neural networks compute, others don't. *Neural Networks*, 2008. doi: 10.1016/j.neunet.2007.12.010.

[227] Gualtiero Piccinini. Computation in physical systems. *The Stanford Encyclopedia of Philosophy*, 2010.

[228] Gualtiero Piccinini and Andrea Scarantino. Information processing, computation, and cognition. *Journal of Biological Physics*, 2011. doi: 10.1007/s10867-010-9195-3.

[229] Benjamin C Pierce and David N Turner. Pict: A programming language based on the pi-calculus. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[230] Martin Pinzger, Katja Gräfenhain, Patrick Knab, and Harald C. Gall. A tool for visual understanding of source code dependencies. In *IEEE International Conference on Program Comprehension*, 2008. doi: 10.1109/ICPC.2008.23.

[231] Emil Post. Degrees of recursive unsolvability: preliminary report. *Bull. Amer. Math. Soc.*, 54:641–642, 1948.

[232] Marian Boykan Pour-El. The structure of computability in analysis and physical theory: An extension of church's thesis. In E.R Griffor, editor, *Handbook of Computability Theory, Studies in Logic and the Foundations of Mathematics.*, pages 449–471. Elsevier, 1999. doi: 10.1016/S0049-237X(99)80029-9.

[233] Michael Prasse and Peter Rittgen. Why church's thesis still holds. some notes on peter wegner's tracts on interaction and computability. *Computer Journal*, 1998. doi: 10.1093/comjnl/41.6.357.

[234] Zenon Pylyshyn. *Computation and Cognition. Toward a Foundation for Cognitive Science.* MIT Press, 1986.

[235] Thibault Rafaillac. *Améliorer les langages et les bibliothèques logicielles pour programmer l'interaction.* PhD thesis, 2019.

[236] William J. Rapaport. What is a computer? a survey. *Minds and Machines*, 2018. doi: 10.1007/s11023-018-9465-6.

[237] Darrell R Raymond. Reading source code. *Proc. CASCON*, 1991.

[238] Trygve Reenskaug. Thing-model-view-editor - an example from a planning system. technical note, xerox parc, 5 1979.

[239] Wolfgang Reisig. Temporal logic and causality in concurrent systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1988. doi: 10.1007/3-540-50403-6_37.

[240] Steven P. Reiss and Alexander Tarvo. Tool demonstration: The visualizations of code bubbles. In *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013. doi: 10.1109/VISSOFT.2013.6650521.

[241] Steven P. Reiss, Jared N. Bott, and Joseph J. Laviola. Code bubbles: A practical working-set programming environment. In *Proceedings - International Conference on Software Engineering*, 2012. doi: 10.1109/ICSE.2012.6227235.

[242] J. Brendan Ritchie and Gualtiero Piccinini. Computational implementation. In *The Routledge Handbook of the Computational Mind*, 2020. doi: 10.4324/9781315643670-15.

[243] Douglas Taylor Ross, John Erwin Ward, et al. Investigations in computer-aided design for numerically controlled production. 1968.

[244] Welsey Salmon. *Scientific Explanation and the Causal Structure of the World*. Princeton University Press, 1984.

[245] Guido Salvaneschi. What do we really know about data flow languages? In *PLATEAU 2016 - Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, co-located with SPLASH 2016*, 2016. doi: 10.1145/3001878.3001884.

[246] Guido Salvaneschi. What do we really know about data flow languages? In *PLATEAU 2016 - Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, co-located with SPLASH 2016*, 2016. doi: 10.1145/3001878.3001884.

[247] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *Proceedings - International Conference on Software Engineering*, 2016. doi: 10.1145/2884781.2884815.

[248] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *Proceedings - International Conference on Software Engineering*, 2016. doi: 10.1145/2884781.2884815.

[249] Guido Salvaneschi and Mira Mezini. Debugging reactive programming with reactive inspector. In *Proceedings - International Conference on Software Engineering*, 2016. doi: 10.1145/2889160.2893174.

[250] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2014. doi: 10.1145/2635868.2635895.

[251] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, pages 25–36. ACM, 2014. doi: 10.1145/2577080.2577083.

[252] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. Reactive programming: A walkthrough. In *Proceedings - International Conference on Software Engineering*, 2015. doi: 10.1109/ICSE.2015.303.

[253] Matthias Scheutz. Computational versus causal complexity. *Minds and Machines*, 2001. doi: 10.1023/A:1011855915651.

[254] Matthias Scheutz. What it is not to implement a computation: A critical analysis of chalmers' notion of implemention. *Journal of Cognitive Science*, 2012. doi: 10.17791/jcs.2012.13.1.75.

[255] Céline Schlienger, Stéphane Conversy, Stéphane Chatty, Magali Anquetil, and Christophe Mertz. Improving users' comprehension of changes with animation and sound: An empirical assessment. volume 4662, pages 207–220, 9 2007. ISBN 978-3-540-74794-9. doi: 10.1007/978-3-540-74796-3_20.

[256] Kjeld Schmidt and Jørgen Bansler. Computational artifacts: interactive and collaborative computing as an integral feature of work practice. In *COOP 2016: Proceedings of the 12th International Conference on the Design of Cooperative Systems, 23-27 May 2016, Trento, Italy*, pages 21–38, 2016.

[257] Roberto Segala, Rainer Gawlick, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. *Information and Computation*, 1998. ISSN 08905401. doi: 10.1006/inco.1997.2671.

[258] Oron Shagrir. Why we view the brain as a computer. *Synthese*, 153:393–416.

[259] Oron Shagrir. Gödel on turing on computability. In Adam Olszewski, Jan Wolenski, and Robert Janusz, editors, *Church's Thesis After 70 Years*, pages 393–419. De Gruyter, Berlin, Boston, 2006. doi: 10.1515/9783110325461.393.

[260] Oron Shagrir. Computation, implementation, cognition. *Minds and Machines*, 22: 137–148, 2012. doi: 10.1007/s11023-012-9280-4.

[261] Wilfried Sieg. On mind and turing's machines. *Natural Computing*, 2007. doi: 10.1007/s11047-006-9021-9.

[262] Hava T. Siegelmann. Computation beyond the turing limit. *Science*, 1995. doi: 10.1126/science.268.5210.545.

[263] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2006. doi: 10.1145/1181775.1181779.

[264] Gurminder Singh. Requirements for user interface programming languages. In *Languages for developing user interfaces*, pages 115–123, 1992.

[265] Brian Cantwell Smith. *On the Origins of objects*. MIT Press, 1996.

[266] Brian Cantwell Smith. The foundations of computing. In Matthias Scheutz, editor, *Computationlism: New Directions*. MIT Press, 2002.

[267] Robert I. Soare. Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic*, 2009. doi: 10.1016/j.apal.2009.01.008.

[268] Robert Irving Soare. Interactive computing and relativized computability. In *Computability: Turing, Godel, Church, and Beyond*, 2013. doi: 10.7551/mitpress/8009.003.0010.

[269] Ludwig Staiger. Omega-languages. In *Handbook of Formal Languages*, 1997. doi: 10.1007/978-3-642-59126-6_6.

[270] Steven E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 2004.

[271] Lucy A Suchman. *Plans and situated actions: The problem of human-machine communication*. Cambridge university press, 1987.

[272] Matti Tedre. *The Science of Computing: Shaping a Discipline*. Chapman Hall/CRC, 2014. ISBN 1482217694.

[273] Wolfgang Thomas. Automata on infinite objects. In *Formal Models and Semantics*, 1990. doi: 10.1016/b978-0-444-88074-1.50009-3.

[274] Allen B. Tucker. *Computer science handbook, second edition*. 2004. ISBN 9780203494455.

[275] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1937. doi: 10.1112/plms/s2-42.1.230.

[276] Alan Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 1939. doi: 10.1112/plms/s2-45.1.161.

[277] Alan Turing. Computing machinery and intelligence. *Minds*, 59:433–60, 1950. URL `http://www.jstor.org/stable/2251299`.

[278] David Ungar, Henry Lieberman, and Christopher Fry. Debugging and the experience of immediacy. *Communications of the ACM*, 1997. doi: 10.1145/248448.248457.

[279] Jan van Leeuwen and Jiří Wiedermann. The turing machine paradigm in contemporary computing. In *Mathematics Unlimited — 2001 and Beyond*, 2001. doi: 10.1007/978-3-642-56478-9_59.

[280] Adriaan Van Wijngaarden, Barry J Mailloux, John EL Peck, Cornelis HA Koster, Michel Sintzoff, Charles H Lindsey, Lambert GLT Meertens, and Richard G Fisker. *Revised report on the algorithmic language ALGOL 68*. Department of Computing Science, the University of Alberta, 1974.

[281] Bret Victor. Learnable programming–designing a programming system for understanding programs. 9 2012. URL `http://worrydream.com/LearnableProgramming`. Accessed: 2021-08-20.

[282] Dong Bach Vo, Kristina Lazarova, Helen C. Purchase, and Mark McCann. Visual causality: Investigating graph layouts for understanding causal processes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2020. ISBN 9783030542481. doi: 10.1007/978-3-030-54249-8_26.

[283] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 35, 2000. doi: 10.1145/358438.349331.

[284] Peter Wegner. Research paradigms in computer science. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 322–330, Washington, DC, USA, 1976. IEEE Computer Society Press. doi: 10.5555/800253.807694.

[285] Peter Wegner. Interaction as a basis for empirical computer science. *ACM Computing Surveys (CSUR)*, 1995. doi: 10.1145/214037.214092.

[286] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 1997. doi: 10.1145/253769.253801.

[287] Peter Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 1998. doi: 10.1016/S0304-3975(97)00154-0.

[288] Peter Wegner. Ubiquity symposium 'what is computation?': The evolution of computation. *Ubiquity*, 2010. doi: 10.1145/1880066.1883611.

[289] Peter Wegner and Dina Goldin. Interaction as a framework for modeling. 1999. doi: 10.1007/3-540-48854-5_19.

[290] Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 2003. doi: 10.1145/641205.641235.

[291] Peter Wegner and Dina Goldin. Principles of problem solving. *Communications of the ACM*, 2006. doi: 10.1145/1139922.1139942.

[292] Jiří Wiedermann and Jan Van Leeuwen. How we think of computing today. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008. doi: 10.1007/978-3-540-69407-6_61.

[293] Jiří Wiedermann and Jan van Leeuwen. Emergence of a super-turing computational potential in artificial living systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001. doi: 10.1007/3-540-44811-x_5.

[294] Nick Wiggershaus. Why we need an agential theory of implementation. 2022. URL `https://hal.archives-ouvertes.fr/view/index/docid/3792142`.

[295] Glynn Winskel. Introduction to operational semantics. In Michael Garey and Albert Meyer, editors, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[296] Claes Wohlin and Aybüke Aurum. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, 2015. doi: 10.1007/s10664-014-9319-7.

[297] David L. Woods, John M. Wyma, E. William Yund, Timothy J. Herron, and Bruce Reed. Factors influencing the latency of simple reaction time. *Frontiers in Human Neuroscience*, 2015. doi: 10.3389/fnhum.2015.00131.

[298] Robert K. Yin. *Case study research : Design and methods*. 2009. doi: 10.1097/FCH.0b013e31822dda9e.

Appendices

Appendix A

Smala's syntax

At the syntactic level, Smala provides specific operators to define causal links between nodes. For example, the arrow between two nodes specifies that each time the left node is activated, the correct node must be activated (binding).

Listing A.1: Simple causal link

```
Clock cl (500)   // a clock ticking every 500ms
Incr inc         // an increment operator
cl.tick->inc     // increment on each tick
```

The double arrow is a data-flow operator for the propagation of values (connector).

Listing A.2: Simple data-flow

```
Double v (0)
inc.state =:> v // copy the sate of the increment
                // to v on each change
```

More complex control structures exist, such as Finite State Machines (FSM). An FSM consists in declarations of states, followed by declarations of transitions ($state->state(event)$). Each state may include other nodes.

Listing A.3: Finite State Machine

```
FillColor f(0,0,0)
Rectangle r (50, 50, 100, 70, 5, 5)
FSM simple_FSM {
  State idle {
    #000000 =: f.value
  }
  State hover {
    #a0a0a0 =: f.value
```

```
  }
  State pressed {
    #ff0000 =: f.value
  }
  idle->hover (r.enter)
  hover->idle (r.leave)
  hover->pressed (r.press)
  pressed->hover (r.release)
}
```

The SMALA syntax has been designed to facilitate the development of interactive software. Indeed, the arrow-based notation provides a first visualisation of the control flow of an interactive software. The language designers have also reversed the direction of the classical APL assignment operator ($src =: dst$ instead of $dst := src$) to make it consistent with the directions of the other types of flow (binding:$->$ and data-flow/connector:=>).

Appendix B

Interview transcripts

I1

27/08/2020

-En ce moment, je travaille sur un projet, Darius. Ça consiste à digitaliser un aéroport de région. En l'occurrence, c'est Tarbes, et on fait ça en partenariat avec une entreprise : Altys. Le but, c'est de développer une interface qui gère les escales, avec les équipes qui assurent le transit, les ressources, le refuel, le catering etc. C'est un projet d'environ deux ans et demi. Après, il faudra que je te parle de projets plus anciens, comme MoTa. C'est une image radar pour contrôleur seul, qui lui permet d'assigner une route, ou de voir une route assignée par défaut.

-Comment se passe le travail, comment commences-tu à coder ?

-T. est le designer. Il propose le concept de l'IHM via des fichiers SVG (Infinity Designer), présentant différentes phases de l'utilisation ou scénarios d'utilisation. C'est parfois un problème, car ce que propose T., c'est une version en cours d'utilisation, et ce n'est pas forcément ce dont le programmeur a besoin pour coder. Le programmeur, lui, a plutôt besoin de voir regroupées les options qui requièrent des animations, avoir un repère pour gérer les translations, voir rapidement ce qui doit être groupé/dégroupé au moment d'interaction/animation. Donc on n'a pas encore trouvé le process optimal. On fonctionne en faisant des allers-retours au SVG, pour que le designer adapte le fichier aux besoins du codeur. Ensuite, sans interagir avec le designer, il y a une phase plus personnelle, où je fais un fichier SVG propre à la représentation d'une icône. Je me lance ensuite dans le code, sans dessiner à la main, avec une démarche itérative. On vérifie au fur et à mesure que les interactions proposées sont viables. A partir de là, le codeur réfléchit à des composants, partage de fichier avec le designer (les diffs). Il y a tout un travail de structuration, de nettoyage qui permet de faire des classes ou composants qui pourront ensuite être ensuite instanciés dynamiquement, aussi un travail de groupement/dégroupement d'objets. En fait, le designer veut une vision globale, moi je veux plus des icônes.

-Tu as des exemples récents de problème que tu as rencontré dans ton code ?

-Alors j'ai un exemple de problème sans issue (enfin, j'ai dû abandonner l'idée) : un drag&drop devant affecter un ensemble de gens sur une tâche (une tâche sur le parking d'un avion), mais par manque d'architecture, j'ai dû laisser tomber. J'ai eu d'autres difficultés dont je peux te parler. Sur le projet MoTa, par exemple, c'était difficile de passer du mode suggestion de route au mode modification manuelle de la route. Ce n'était pas un problème d'implémentation, mais un problème de workflow et de gestion des FSM. Par exemple, rester dans un état sans transition sortante. J'ai un collègue qui d'ailleurs utilisaient les FSM dessinées à la main pour mettre à plat le problème. Après en Python, Qt, j'ai eu des problèmes de SVG, dès que je veux redécouper le fichier pour en changer un bout. Ah ! et

sur le projet Darius, un problème, c'était l'ordre, classer les vignettes correspondant à un vol, dans l'ordre de leur futur départ du parking. C'était un problème de typage automatique, pour classer des heures de départs : des entiers ou des chaînes de caractères. Pour trouver l'origine du problème, j'ai dû tracer dans le code, tester sur des critères différents pour identifier l'origine de l'erreur. Je devais aussi galérer sur la taille de la frame : dans la Native Action. L'ordre des déclarations change la taille des lignes au lieu d'obtenir une frame qui s'adapte à la taille de la fenêtre. Ce n'est pas clair si c'est un problème qui dépend de l'exécution. Avec la dynamicité, j'ai d'autres soucis. Quand on lance l'application, il faut un timer d'une seconde. On créé une application vide puis après expiration du timer, le fichier est lancé, c'est-à-dire que toutes les structures sont créées. Sinon on a des problèmes de parentage au démarrage, et des problèmes de dynamité. Il faut créer un à un les panneaux vides et find pour mettre les éléments dedans. On voudrait un on-start. Dans le projet MoTa que tu vois, je peux te montrer les endroits où j'ai dû passer du temps sur la gestion de la destruction dynamique, on veut désactiver tous les bindings avant de supprimer l'objet graphique Pour les FSM, comme ici, je dois souvent vérifier et m'en remets à l'utilisation de print dans le programme pour vérifier qu'on est dans le bon état ; par exemple quand j'ai des objets qui bougent sans qu'on sache l'événement qui l'a déclenché. L'utilisation des masques en plus créé des problèmes quand on définit des transitions, ça fait des objets qui se masquent les uns les autres. Cela complique l'écriture des zones à clic. Quand il y a plusieurs FSM imbriquées, c'est pénible à écrire. Mais les erreurs dans l'écriture des transitions, si on en a oublié une, sautent aux yeux. Il y a des transitions qu'on est obligé d'expliciter et dont on aimerait parfois se passer, mais c'est un parti qui à l'inverse a ses inconvénients. (Le participant continue à montrer des portions de code et à naviguer dans l'IDE). Après ici, tu vois, quand on veut faire plusieurs actions sur un même objet (click, click long et double click; ou un drag&drop qui puisse remplacer une valeur par une autre, et être elle-même annulée par un clic), ça requiert des FSM complexes. La solution, c'est de réduire le nombre d'action sur le même objet, et faire de l'essai-erreur. Pouvoir mettre à plat le problème de l'imbrication de FSM aiderait. Après, je pense à un problème propre à la phase de prototypage : on fait surtout des trucs purement graphiques, et on ne stocke pas forcément les noms et les tâches. On peut bidouiller des trucs moches comme générer des flux JSON qui stocke ces données. Ici, on répond d'abord à un besoin (proposer un graphisme et des interactions pour un futur produit), le critère de propreté viendrait après. Je suis contraint de faire de l'aller-retour entre le code djnn et un fichier JSON pour retrouver point par point le bon path. C'est un problème de structure, il faut reparcourir le fichier djnn , debugger, avec des find et dump, c'est pas toujours parfait. En particulier, il y a les duplications de nom dans les textfields (textfield.textfield.text) des SVG. Je dois faire un dump de l'arbre : c'est lourd à manipuler. Les paths en général, ce sont des erreurs à tracer, comme l'identifiant en Inkscape (il suit la spec' de XML, notamment la règle de l'identifiant unique) qui force un nom quand on a réutilisé le même. C'est une règle qui n'a pas son utilité, vu qu'on utilise l'arborescence pour retrouver un nom, donc les duplications de noms ne posent pas problème. Je sais que je vais avoir des problèmes avec la gestion des sources multiples, quand il faudra rajouter du multi touch généralisé ou un nouvel inter acteur : dans ce cas, il faudra

rajouter partout dans le code des Spike, pour mettre à jour tous les press. Avec MoTa aussi, il faut que je t'explique un truc. On a l'utilisateur et les capteurs qui se substituent au radar et font de la reconnaissance d'objets. Donc on a d'un côté toutes les secondes les updates des informations du capteurs, et en même temps il faut que l'utilisateur ait le temps de percevoir et comprendre l'information.

I2

16/01/2021

-Il faudra que je te parle du projet dans ma thèse, parce que je m'en souviens bien. La thèse impliquait de l'aéro dans le sens où on a voulu utiliser des concepts développés par le labo sur les systèmes ATC pour les appliquer dans un hôpital, en salle de chirurgie. On faisait un parallèle entre la gestion d'un hôpital et sa suite chirurgical, avec les méthodes ATC. Donc je ciblais les interfaces ATC, pour voir si on pouvait développer le même type d'interface pour le staff des salles de chirurgies. Les strips, les interactions direct sur surfaces multi-touch, les aspects collaboratifs. Aujourd'hui je suis entreprise chez X. pour le contrôle aérien, dans une équipe qui s'occupe des systèmes pour les tours de contrôle. Cela implique moins d'IHM, mais plutôt de s'occuper des systèmes qui ramènent l'info (radars etc.), et de la rédaction de spécification pour les différents clients dans le monde. On a des grandes grandes équipes, moi je suis dans une sous-équipe d'une trentaines de personnes. Des équipes qui font de tout. X. fait depuis le développement des radars eux-mêmes jusqu'à la conception de l'IATS (tour de contrôle intégrée). Pour la constitution d'une équipe, tu as les software engineers qui développent et debug le code, les system engineers (moi) qui font les spécifications, répondent aux besoins des clients, toujours sur la partie produit de base et comment on le fait évoluer, et l es "projects" qui bossent sur comment on adapte le produit aux différents clients.
Aujourd'hui, je ne code plus, mais lis du code pour comprendre les problèmes, et faire de la documentation.

-Que ce soit sur ton projet actuel en entreprise ou dans tes souvenirs frais de thèse, tu as des difficultés dont tu aimerais parler, des bug récurrents ?

-J'ai beaucoup développé "On-board", un écran 84K microsoft, développé une interface pour suivre les patients qui arrivent dans une suite chirurgicale (voir ça comme des avions qui arrivent à l'aéroport).

C'était une architecture en Java avec des classes et tout, et ensuite l'interface, sur laquelle je voulais permettre beaucoup d'interactions directs, avec des post-its, des stickers. Ce qu'on essayait de faire, c'était de développer des interactions fines, des comportements d'objet très naturels. Donc la difficulté c'était de faire quelque chose qui avait l'air simple, mais derrière c'était très compliqué de coder au pixel près pour avoir exactement la bonne animation. Pour coder quelque quelque chose de beau et simple, il faut derrière quelque chose de compliqué. Par exemple, un tableau vertical avec des bandelettes comme des strips digitaux de contrôleur aérien, et on peut pouvoir les déplacer. Mais on veut qu'il puisse venir se glisser entre deux autres strips. Ca n'a pas de grande finalité en soi, mais on veut rendre disponible ce genre d'animation, en faisant l'hypothèse qu'elle pourra servir aux utilisateurs. Là on veut assurer la fluidité, on ne veut pas que tout bouge d'un coup, on veut une harmonie. Il faut trouver les bons timings, les objets doivent bouger à une certaine vitesse, penser leur changement de taille, leur déplacement et direction. Les gros problèmes sinon, c'était sur l'écriture. On voulait pouvoir écrire sur les strips comme sur du papier. C'était déjà en train d'être amélioré par Apple et microsoft. C'est challenging de coder de l'écriture très fine. Ce n'est donc peut-être pas d'actualité. Ecrire sur une surface multitouch pose des problèmes de hardware/capteurs: il faut que la source (en grosl'écran qui reçoit les inputs) doit avoir une grande finesse, on ne peut pas avoir un point capté tous les cinq centimètres. Il faut une matrice de points très fine pour rendre une écriture naturelle. Et ensuite il y a l'algorithme derrière: veut-on des lignes droites entre chaque coordonnées? Mais c'est insuffisant. Il faut adoucir les courbes. On veut aussi du feed-back direct, la difficulté, c'est le temps-réel. Il faut donc notamment anticiper les courbes.

-On peut revenir sur les questions de timing ?

-J'étais donc focalisée sur le retour utilisateur. Choisir le temps, les tailles: c'est tout visuel. C'est un peu long ça ne fait pas naturel, alors je change. J'ajustais en fonction du retour que j'avais moi. J'ai essayé de chercher des règles, des publications qui disent que pour une interaction de tel type, il faut telle vitesse de déplacement. Donc c'était au cas par cas, moi qui estimais. Généralement, c'était suffisant, mais ça prenait beaucoup de mon temps: changer d'une milliseconde, et rejouer rejouer jusqu'à ce que ça me plaise. C'est pas du bug, c'est une finesse d'interaction que tu veux développer. D'abord tu mets telle vitesse, tu simules, tu vois comment ça se passe, ca va trop vite, pas assez vite, tu retournes au code et tu ajustes.

-Tu pourrais imaginer des outils qui auraient pu t'aider ?

-Des guidelines sur un objet avec telles propriétés, et ce qui est optimum. Ca existe un peu, par exemple pour un bouton à atteindre, sur la taille qu'il doit avoir pour que ce soit utilisable. Travailler avec des objets réels, manipuler des objets physiques et tu vois comment ça se passe, comment ça bouge.

-Comment tu travaillais d'ailleurs, une fois l'idée de l'interaction en tête ?

-La technique de base, tu dessines sur du papier, tu travailles direct avec les utilisateurs finaux, des mock-ups, tu montres ces mock-ups aux utilisateurs. Tu fais une enquête pour voir comment ça répond aux besoins sur une grande feuille de papier. Ensuite, tu peux coder pour une surface tactile et tu refais le test avec l'utilisateur, voire si ça bloque, s'il y a des choses que les utilisateurs ne peuvent plus faire, si ça les aide à faire moins d'erreur. Ensuite, forcément tu fais des méthodes de software, tu dessines des classes, ton tableau de classes. Après, le rêve du codeur, c'est de coder son interaction, la lancer et ça marche. On aimerait un tableau qui dise que c'est là que ça merde. Par exemple, tu as une interface avec trois post-its et tu as un post-it qui merde. Tu cliques sur le post-it et boum ça t'ouvre le morceau de code avec le post-it, avec d'autres aides pour comprendre où ça ne marche pas, on te dit "c'est là", et ces lignes-là sont suspectes. Quand tu as des milliers de lignes de code et qu'un truc ne se passe pas comme tu veux, ce n'est pas forcément là où tu travailles qu'il y a l'erreur. Si tu as une espèce d'assistant qui peut te dire où ça se passe. Et ca existe, Visual Studio ou Eclipse debugger. Mais là ces debuggers t'indiquent les erreurs. Le problème, c'est que pour une interface, tout peut bien se passer mais visuellement ça ne se passe pas comme tu veux. Visuellement un truc qui plante. Mais c'est très difficile, parce que c'est humain. Je me souviens d'un outil qui avait été développé sympa, pour les machines à états, SwingStates. Je ne l'ai pas utilisé mais je trouvais que c'était un bon concept. Savoir dans quel état tu es de l'animation, tu peux faire tourner l'outil en même temps que ton code. Et tu peux voir que le post-it reste dans un état neutre alors qu'on veut qu'il soit dans l'état "on peut écrire dessus". On comprend alors qu'une transition n'a pas eu lieu. Valable surtout si tu as 70 animations.

-A part SwingStates pour les machines à états, tu verrais autre chose pour aider à comprendre le code interactif?

-Ca dépend quelle vue d'esprit, quel concept tu veux utiliser quand tu as les objets sur ton interface. Moi j'ai travaillé avec les machines à état parce que ça allait très bien avec ce que je faisais. Par contre avec une page web, ca marcherait pas aussi bien. Dans tous les cas, la réponse à la question ne peut pas être universelle. Ca dépend du type d'interface que tu veux déployer. Mais l'idée d'une interface qui évolue en même que l'interface que tu développes, ça c'est un concept qui peut être appliqué dans beaucoup de cas. Il y a peut-être besoin de catégoriser le type d'interface qu'on peut être amené à développer. C'est vrai que le debugger est le seul outil de dispo pour trouver les bugs "ordinateurs", l'autre outil ensuite c'est les yeux, ta perception. Les bugs "haptiques", à part toi même être humain, tu n'as pas d'outils pour analyser ces choses-là. On peut imaginer un algorithme qui tourne et qui évalue par exemple, disons je veux que mes post-its aient un comportement d'objets physiques (loi de la gravité, du vent etc…on peut imagine une situation), l'algorithme détecte par exemple qu'un post-its ne fait pas de déplacer un autre post-it alors qu'il est censé le pousser. On créé une intelligence, un assistant qui tourne en parallèle et que tu entraînes à penser comme toi.

I3

21/01/2021

-Après mon_master en IHM, je suis partie chez X., un site de vente de billets d'avion, et j'ai bossé pour le développement website pour la compagnie aérienne S. On est une grosse boîte multinationale, avec des équipes dans le monde entier. Moi je bosse pour S., sur le booking flow. Pour faire ce site web, on est un centre consacré à ça, on est 5 équipes dessus, chacune de 5 à 10 personnes. Moi je suis développeuse et Scrum Master. On prend des petites parties du site web qu'on améliore. Par exemple, la partie où on sélectionne les passagers, les types de passagers, leur nombre, travailler sur la génération du mail pour les envoyer dans différentes langues (arabe notamment, écriture de droite à gauche) et les rendre compatibles. Donc on touche progressivement à des briques du site pour le rendre global. C'est un projet d'environ 3 ans, gros projet, faut refaire tout un site web.

-Comment tu travailles quand tu codes ?

-On a une user story pour commencer quand il s'agit d'introduire une nouvelle brique. On a un PO (product owner) qui est censé te préparer tout le travail en avance, qui va normalement discuter avec la personne qui gère le produit, pour savoir comment l'intégrer. Les mockups ont déjà été faits; on utilise zeppelin: des mockups très haute fidélité, quasiment déjà un site interactif; quand tu cliques sur un élément, tu as toute sa description déjà en code (html). Le rendu est déjà là. Si on a des questions sur la user story, on demande des précisions (pour le comportement par exemple). Ensuite on se demande comment on va faire les choses; ça a lieu pendant des meetings avec les autres développeurs. En général, c'est tellement clair, qu'on n'a pas besoin de ces meetings. Mais j'ai un exemple où cela a été nécessaire. On a une site map, et en fonction du nombre de passagers, on va mettre des tabs en haut qui vont jusqu'à neuf passagers, mais cela ne tient pas sur une ligne. Donc le comportement à la base, c'était des chevrons, mais on nous a demandé que ça devienne swipable sur mobile. Nous on a avait une material tab, donc ne rend pas possible le swipe. Donc il fallait trouver une solution pour rendre ça "swipable". On a fait une étude (regarder sur d'autres sites web, et demander à d'autres équipes les solutions trouvées) et ensuite on choisit la meilleure solution. Il a fallu changer les propriétés CSS.

-Tu as des animations à coder qui ont pris du temps, posé des problèmes particuliers ?

-Pendant mon stage, toujours dans le milieu aérien chez X., je bossais sur un logiciel pour superviser tous les kiosques d'X. On ne voyait jamais très bien la proposition de kiosque hors service. On m'a proposé

de moderniser l'appui, de faire un dashboard avec une roue offrant une vue de l'état des kiosques (avec trois états, vert, orange et rouge). En cliquant sur la roue, on pouvait zoomer sur différents pays. Comportement intéressant, mais difficile à faire marcher, à cause du formatage des données pour l'API. Je partais avec une base de données de 20 000 kiosques, en SQL, transformées en backend en objet JSON. Donc on récupérait un énorme JSON. La solution consistait non plus à partir des 20 000 postes, mais à partir d'un kiosque, essayer d'en afficher un, puis deux avec deux états différents, et progressivement agrandir la base de données, en voyant à partir de quand ça cassait. J'y suis allée par incrément. Ce qui aurait aidé, c'est d'avoir un exemple écrit.

-Et plus récemment ?

-Les problèmes qui se posent souvent, ce n'est pas forcément des problèmes de code, mais c'est lié au fait qu'on travaille avec des librairies créées par d'autres personnes et donc on met à jour régulièrement et parfois la librairie a subi des modifications qui ensuite cassent tout dans notre application. Donc c'est la recherche de qui a touché quoi et comment le résoudre. Mais pour moi, ce n'est pas un gros problème de code et d'interactions. L'email, ça a été une vraie galère. Ce qui est intéressant, c'est qu'à chaque fois que tu codes, tu ne sais pas si tu vas obtenir le même résultat chez tous les clients. Safari, Firefox etc..tu en testes 4-5 et ensuite tu décides que ça devrait marcher aussi pour les versions supérieures. Outlook, Gmail, Gmail mobile, Hotmail. Un sacré panel, et leur logiciel de rendu ne sont pas du tout les mêmes. Par exemple, Gmail va se comporter pas mal comme du JavaScript, Outlook va se servir du rendu de Word. Donc le HTML va être interprété en tableau word, ce qui casse tout.

-Tu as des souvenirs pendant ton master IHM?

-Oui, j'avais projet PIR que j'ai beaucoup aimé, "Where's My Drone?". Des lunettes en réalité augmentée, pour qu'un pilote de drone puisse suivre le drone à distance et essayer d'améliorer la façon dont il suit le drone pour ne pas le perdre. Il fallait gérer l'affichage des vitesses, alertes, indication batterie, direction du drone. Le projet, c'était de trouver un nouvel axe d'amélioration, suivre le drone plus précisément. On a trouvé dans l'état de l'art une idée intéressante de "mini-carte" qu'on voulait rendre en 3D pour suivre également l'altitude du drone. Un point rouge (ta position), un triangle (le champ de vision) et un point bleu avec un trait pour représenter le drone et son altitude. Codé avec Unity/C Sharp. J'avais énormément de mal à voir si ce que je codais correspondais au vrai comportement dur drone dans la vraie vie. Le but, c'était aussi de le faire voler avec d'autres drones, et donc c'était difficile de rendre les positions des drones les uns par rapport aux autres. Ou trouver un marqueur pour indiquer que le drone approche du ras du sol.

-Comment tu réajustais pour matcher au comportement du drone ?

-Beaucoup à tâtons. Cela aurait été sympa d'avoir un logiciel pour simuler le comportement du drone, pour voir si l'interface répond au même moment au changement de comportement. C'est vraiment ce qui manquait. Ce qui a été compliqué, c'était le re-calcul des angles

entre le drone et le triangle du champ de vision en fonction du déplacement de l'utilisateur. Je peux te parler d'un autre projet au labo de Bristol: le but, c'était de rendre les interactions physiques, comment on rend l'impression qu'on presse un objet… Une fois qu'on attrape un objet, comment créer des points de pression. Aussi un projet en 3D, avec Unité/C Sharp. L'idée, c'était de prendre des formes, avoir un modèle de mai 3D et le projeter sur l'objet, calculer par projection les surfaces d'interactions avec Unity, données 3D qu'on peut ensuite exporter en python. Les points de pression en 3D. Mon truc c'était vraiment ciblé les empreintes digitales, savoir exactement quelle partie de la main était rentrée en contact. Beaucoup d'allers-retours à la documentation sur le site et collaboration avec l'équipe, tâtonnements. Un autre projet du master sinon, c'étaot A R++. On bossait à quatre dessus. Un logiciel de statistique à améliorer. On a choisi de faire une version pour tablette, qui permettait de se connecter au logiciel, de récupérer des données, graphiques et de directement interagir via la tablette sur le PC pour projeter les données du logiciel sur un écran principal. Donc permettre une présentation powerpoint où on puisse directement aller interagir avec la tablette et modifier les graphiques. Un problème : on avait énormément de mal à réussir à tracer quelque chose sur la zone de dessin et ensuite en retenir une image pour la réexpédier dans le réseau (la reconnaissance de forme sur la zone de dessin) et l'envoyer à l'écran secondaire. Très mauvaise superposition des deux images: du cercle dessiné et du png dessus, les superposer, en faire une seule image pour l'envoyer sur l'autre écran. La solution: redéfinir l'axe des 0 de chaque dessin. C'était un problème de reconnaissance des actions aussi. La multimodalité pas évidente, ici la reconnaissance de symboles. Dans nos projets actuels, on a des problèmes similaires, où une action n'est pas reconnue sur certains devices. Genre un swipe qui soudain fait tout plante sur un device particulier. Et le problème qu'on a là, c'est d'essayer de reproduire, qu'est-ce qui est interprété? Cela arrive avec l'IPhone X. Le code n'est pas ciblé et ne marche pas sur une certaine version. Il y a des events du touch qu'il faut préciser par exemple: par exemple préciser que pour certain touch, il ne doit rien se passer, car sinon on risque de ne pas du tout avoir ce qu'on veut. Par exemple, on était obligé d'interdire la réaction à des touch ou drag de l'utilisateur, dans le cas des pop-ups. Ici ce serait intéressant d'avoir un simulateur de gestes et d'applications, simuler des mouvements aléatoires, pour qu'on nous fasse remonter les problèmes tout de suite avec ces tests.

-Tu as des idées d'outils qui pourraient aider ?

-Quand tu codes, tu aimes savoir quand il y a des erreurs, quand un truc ne marche pas comme tu veux. Par exemple, quand on code sur Html, on a les erreurs de la console. Et ce qui pourrait être intéressant, ce serait du step by step. Ce qu'on arrive à faire c'est définir des points clés; on sait qu'il y a eu un problème à tel endroit, et nous quand on débeug, on demande au debugger de s'arrêter là. Mais on n'a pas de flow, on n'a pas l'idée de prendre des actions successives, des scénarios en présélectionnant des actions par défaut. Et on veut que le step-by-step soit dans le scénario, pas dans l'exécution du code. Pouvoir cibler le moment où on sait que l'appui est à deux doigts de crasher. Avoir une application où tu peux passer d'un état à l'autre grâce à un scénario prédéfini, et avoir comme une barre de progression, une map en 3D éclatée ou un réseau, pour pouvoir

sélectionné un état qu'on veut voir. Un truc pour naviguer dans l'application avec un outil, on verrait par exemple que tel scénario nous fait atterrir dans un state qui n'est pas bon. On verrait beaucoup plus vite où ça casse. Etre capable d'avoir un lien complet entre les scénarios que t'as et pouvoir circuler de l'un à l'autre, pour passer d'un état à un autre, comme une machine à états. Par exemple pour le booking flow chez X., on veut casser le côté linéaire, et laisser l'utilisateur pouvoir se déplacer dans l'application avant le paiement; et dans ces cas là, on veut pouvoir vérifier si la seat map va crasher à un moment à cause d'un certain état. Quitte à avoir des vues où on pourrait rentrer dans un certain état, zoomer dedans, et avoir une granularité (le choix des passager, sièges etc).

I4

28/01/21

-Après le master, j'ai été dans le développement pour application mobile, mais je ne sais pas si ce que je faisais va être intéressant pour toi, parce que depuis il y a beaucoup d'outils qui ont été développé pour faire du prototype rapide de comportements d'IHM, notamment pour iOS avec Playground mais que je connais pas bien. Donc les problèmes dont je vais te parler seront peut-être dépassés et facilement solvables aujourd'hui. J'étais dans une filiale de X., j'avais un rôle proche de l'IHM, identifier les besoins des ingénieurs des bancs d'essai, les aider à mieux concevoir et développer les outils qu'ils utilisent pour monitorer les essais qu'ils réalisaient, un rôle transversal. Depuis 2020, j'ai eu un rôle de product owner dans une filiale de C. Je ne faisais pas que de la spec, pas de code en tant que tel, mais partie technique, j'étais amené à lire du code. On devrait programmer un système lié à la protection contre braquage de bateaux sur les eaux internationales, à la fois le front et le back (où je suis particulièrement en charge de ce qui concerne les échanges entre les balises jusqu'à l'application). Quand j'étais dans le mobile, pour développer le front, c'était de l'Objectif C, le back c'était du Java. On était une équipe Scrum Agile, avec en moyenne deux trois autres développeurs et un designer, et un product owner et un testeur. Des projets de quelques semaines à trois-quatre mois, souvent des sous-traitances pour des start-ups. Notre façon de travailler… on travaillait en deux temps; d'abord le designer itère avec le client et le product owner, pour faire des maquettes moyenne fidélité, donner une idée du flow entre les écrans. Ensuite, le designer faisait des prototypes haute fidélité au pixel près, avec exactement le type de padding.

-Toi, tu avais une manière particulière de travailler une fois les maquettes du designer reçues ?

-Parfois on discutait avec le designer. On faisait des séances de sprint planification. Chacun recevait un ticket avec un écran ou une description à coder.

-Des problèmes récurrents ou marquants lors de l'expérience dans les applications mobiles?

-Alors il y a eu un problème particulièrement difficile lors d'un projet d'application qui devait permettre de réserver des services offerts par des particuliers qui ont des compétences (bricolage, plomberie). L'application centrée sur Londres montraient les jobs et le tarifs, à la fois les offres et les demandes (avec date et horaires indiqués). En fonction de la demande, un système d'enchères: si un particulier accepte le job mais plus tard que l'horaire demandé, le prix monte. Et le statut de ces offres étaient très dur à gérer. En gros, il faut idéalement des FSM mais dans la réalité t'as pas le temps, donc tu codes à la va-vite, tu rajoutes des champs dans des tables, tu te retrouves avec des booléns, tu rajoutes des statuts. La notion de condition et comment tu l'implémentes de manière certaine, c'est difficile. On n'a pas toutes les transitions certaines. Ca marche pour des choses simples, pour faciliter des preuves formelles, mais plus dur sur des très grosses applications où il y a 50 transitions et un tas de vues modales. C'est trop chronophage et matche pas avec la vitesse à laquelle il faut développer ces applications. Mais c'est un problème quand on rajoute des features au fur et à mesure. Ca devient difficile de bloquer l'accès à certaines fonctionnalités pour l'utilisateur par exemple, qu'on aimerait interdire dans certaines configurations. Ou tu as un utilisateur qui doit avoir plus de privilèges qu'un autre, mais c'est difficile à expliciter. Donc on fait du testing à mort. Mais écrire et faire c'est scénarios de test, c'est très chronophage.

-Pour ce pb de statut, est-ce que tu aurais imaginé un outil qui t'aurait aidé ?

-Oui, un outil qui aurait permis de réaliser une FSM facilement. Pouvoir créer cette machine aussi complexe soit-elle, sans perdre du temps. Pour moi et c'est prouvé, c'est la meilleure méthode mais on veut que ça marche au-delà des systèmes critiques relativement simples avec quelques boutons. On voudrait ça aussi aussi pour des applications B2C. Une machine à état qu'on puisse créer de manière itérative, parce qu'on veut rajouter des états au fur et à mesure qu'on développe. En général tu sais ce que tu veux coder en termes d'animation et c'est assez facile à réaliser, de rajouter des effets, surtout sur iOS, ça se paramètre facilement. Le SDK de la plateforme te cache le travail. Pour moi le problème, c'est surtout cette notion d'état. Est-ce que j'ai toujours le droit d'aller là etc. ? Ce ne serait pas un outil qui génère automatique du code tout d'un coup. C'est plutôt de permettre d'itérer.
Un autre problème que j'ai en tête par rapport à ce que je fais maintenant sinon, c'est les questions de performance. Là je bosse sur des systèmes où il y a beaucoup de cartographies, et quand on développe, on commence qu'avec quelques points. Et quand ça passe en

production, là tout d'un coup on télécharge beaucoup de données. Et les développeurs quand il réalise les interfaces n'ont pas le même set de données et n'ont pas les mêmes contraintes côté performance. Et des trucs qui marchent super bien en phase de développement, ne marchent plus pareil du tout en phase de production. La notion de charge de données n'a pas été suffisamment prise en compte. Mais ce n'est pas si simple non plus de développer avec des gros jeux de données, ce serait trop lent. Pour la charge de données, je n'ai pas d'idée d'outil. Lié aussi à ce delta entre la phase de développement et phase de production, il y la question du débit internet. La partie back des applis qui fonctionne avec internet, on part du postulat qu'on a une bonne connexion, parce qu'on est Europe. Mais ce n'est pas le cas en Asie. Donc une application mobile n'est pas testée sur du Edge, comment je fais la reprise sur erreur, est-ce que je stocke en local, quand je resynchronise avec le back…etc? Donc développer pour des pays émergents posent des difficultés au niveau du test. Mais ca existe déjà un peu ce genre d'outil pour simuler une applis avec du Edge.
Un autre problème, c'est la relation entre développeur front et développeur back. Parfois les maquettes front sont prêtes et les développeurs back n'ont pas encore commencé, donc les front vont commencer à créer les écrans mais les 3/4 des données qu'ils affichent sont censés venir du backend. Donc comment tu mets en place une signature d'API qui permette au développeur front de développer sans pour autant demander au développeur back d'aller plus vite que la lumière? Un outil qu'on utilise, recourir à des faux jeux de données. Plus général après: concilier les business units (commerciaux, responsables produits, designers) et les équipes techniques qui mettent en oeuvre le code. Ca bloque le processus itératif. Ca serait bien qu'il y ait des outils qui ne soient pas que orientés développement versus design, plutôt un outil qui puisse connecter les deux mondes. Le développeur a parfois besoin d'avoir plus la main, surtout s'il a justement une formation facteurs humains et IHM.

-Comment décrirais-tu tes stratégies de debug?

-Des points d'arrêt là où on pense que ça plante. Et une fois qu'est isolée la zone où ça plante, on peut aussi rajouter des logs. Il y a d'autres outils fournis par les IDE, des graphes et tout. Mais c'est vrai qu'il y a un manque de connaissance sur les potentialités des IDE. Très peu des fonctionnalités sont utilisées. Je n'ai par exemple jamais utilisé de graphes pour debugger.

I5

09/02/21

-Alors moi mon truc, c'est des projets de recherche, créer des interactions qui n'existent pas trop. On commence par Candifly si tu veux, j'ai des problèmes dessus en ce moment. Ce qui est délicat dans ce projet, c'est de faire parler tout le monde, parce qu'on a des Arduino avec des capteurs, l'ordi et un drone, donc on veut récupérer des infos physiques, de capteurs et les balancer par message à un drone et on veut récupérer les infos du drone et potentiellement afficher des trucs sur l'Arduino. Du coup, ce qui est galère, c'est de faire parler tout le monde, parce que personne ne parle la même langue. Arduino en port série, ça envoie du binaire sur USB, avec une boucle qui tourne tout le temps; c'est pas événementiel, c'est plutôt une boucle while et tu fais des SLIP, ce qui bloque tout. Donc la stratégie pour m'en sortir, parce que je préfère travailler avec des événements et j'utilise les signaux de Qt. Je ne veux pas que mon interface freeze à chaque fois qu'il se passe quelque chose. Il faut donc appeler des événements, s'abonner et tout envoyer par signaux. Parce que si tu dors, ça freeze tout. Faut faire super gaffe et c'est chiant. Donc pour utiliser un Arduino, je créé un thread, et lui il tourne en tâche de fond et tu lis sur le port série. Et je le démarre (description du code à l'écran). Et je décode ces données que je mets dans un tableau. Et si jamais des états ont changé, j'émets des clicks. Ca tourne super vite et tout le temps, et je compare les valeurs reçues aux valeurs d'avant et s'il y a un changement, j'émets le click. Je n'ai pas besoin de balancer deux cent fois la même info, donc je gère ça en stockant les données. Je démarre un timer, et quand tu arrives à la fin du temps, utilise la méthode "process". Ca me sert de fréquence de mise à jour. Et ce truc là "process", il va lire dans mon tableau de données. J'ai pas besoin d'envoyer un message réseau à mon drone toutes les millisecondes. Donc j'ai besoin d'une limite en fréquence et c'est chiant à faire. Dans d'autres langages je sais le faire, en déclarant une limite de vitesse. Il y a des opérateurs pour ça, pour mettre les données dans une queue, et tu lui dis que c'est le dernier arrivé que tu fais sortir ou le premier arrivé que tu dépiles; dans le dernier cas, tu as des données pas très fraîches. Donc je fais des tests, je vérifie que ça ait changé, c'est que si au moins une des valeurs des capteurs est différente que j'envoie la donnée. Après quand je fais du streaming, je balance tout le temps, pas sur un mode événementiel. Pour faire un value change en gros. Le drone, par exemple pour les Crazifly, j'utilise des librairies, je m'abonne à des événements. J'utilise des fonctions et des messages radio. Lui aussi il a des limites en terme de messages qu'il peut recevoir par seconde. Donc au début j'en envoyais trop vite, il en ratait plein, ça servait à rien.

-Et pour le choix de la vitesse, comment tu fais ?

-Alors c'est du tâtonnement. Après y a une spéc' et puis il y a réalité aussi: plus c'est loin plus ça met du temps, tu peux pas faire n'importe quoi. Le truc chiant là-dedans c'est de faire parler tous ces trucs ensemble, et d'être sûr, malgré les spéc' et tout, que tu

ne vas pas envoyer trop de messages, que tu vas faire matcher un peu tout. C'est pas facile. En terme de visu, ce que je voudrais peut-être, c'est des compteurs, pour me rendre compte quand ça tourne à quel point ça presse là-dedans, et combien de messages sont utilisés, peut-être une petite jauge, ou une moyenne de messages qui passe par seconde dans un biding, ça ça serait trop cool. Parce que moi à la fin, je vais dire "init le drone", puis après "si le drone il existe, alors je m'abonne à sa vitesse, batterie etc ., si je change les valeurs de vitesse, abonne-toi". Donc je reçois les données de l'Arduino, les affiche dans l'IHM et si j'ai un drone, je dis "traite ce mouvement". Quand tu codes c'est facile, tu prends la source, tu connectes et tu l'affiches, mais tu sais pas si c'est pertinent la vitesse, les flux de données, la quantité que tu balances. Et donc dès fois tu dis 100ms, bon après ça va j'ai de l'expérience, donc je sais un peu. Je préférerais me rendre compte de combien de messages passent et à quelle fréquence c'est appelé. Et de pouvoir voir également pendant un temps les valeurs, pour voir si c'est tout le temps les mêmes ou pas. S'il n'y a presque rien qui change, tu ne veux pas balancer de données tant que tu ne t'es pas écarté d'une certaine distance. Mais ça dépend des trucs, c'est un peu comme un filtre passe-bas. Et tout ce genre de truc, conditionner le signal dans les événements, c'est faisable mais pour voir la gueule que ça a, c'est difficile. Y a un truc qui existe et qui est pas mal, c'est "1 € Filter". Ils ont fait une visu, mais ça ne marche pas avec tout. C'est un filtre pour les inputs, mais plutôt pour la position, pour empêcher que ça bouge trop. En gros, c'est pour lisser la position de la souris. Tu vois le signal. Ce n'est pas la fréquence d'arrivée des données qui est affichée, mais c'est un exemple de visu qui permet de comparer des trucs, le choix de vitesse et de bruit sur ton capteur. Enfin voilà, j'aimerais quelque chose comme ça mais pour réaliser la pression temporelle, gérer la performance. En électronique ça a un nom, c'est l'impédance d'entrée et de sortie, la charge du truc. Y a des trucs pour loger ou tu peux faire des prints, mais faut compter, faut faire des diffs toi-même. Alors des petites jauges où quand tu cliques t'as du détail, ça pourrait être sympa.
On peut parler d'un problème d'animation aussi puisque ça t'intéresse. BDR ça te parle? C'est une application pour des contrôleurs aérien, pour des avions qui a un moment risquent de passer un peu trop près l'un de l'autre. On appelle ça le rapprochement. Il faut jamais qu'ils soient en-dessous de 3000 nautical miles. Et donc l'outil doit te dire "dans 36 sec, l'avion va être à 3 nautiques". T'as cette jauge qui descend, plus elle descend, plus le danger approche et la couleur change. Et y a un peu de son "radar" et une animation en même temps que le son. Il y a une autre animation pour le ballotage. C'est-à-dire qu'à la limite d'une situation de rapprochement, pour éviter qu'avec l'algo ça change à la seconde "ah oui, ah non"; pour ça, on essaye des durées plus longues. En gros, quand il te dit "créé un strip", tu affiches le strip avec son opacité, et quand il faut le détruire, on ne détruit pas le strip tout de suite (pour ne pas qu'il clignote en gros), et on baisse l'opacité dans le temps. Et ce temps il faut le régler. Et en général ce qui est bien, c'est un temps plus long que celui qu'il faut pour une création-disparition. Donc j'ai comparé plusieurs temps. C'est un peu empirique. Parce que si ça se met à clignoter, tu ne sais plus si c'est un nouvel avion ou pas. Tandis que si c'est juste l'opacité qui varie. Donc c'est pas si simple à coder ces trucs: le lien audio vidéo et chiant (je vais te montrer le code), et le temps des animations aussi.

-Et justement pour les temps à ajuster pour l'opacité et éviter le clignotement du strip aux valeurs limite pour un "rapprochement" entre avions, il y aurait un outil qui aurait pu t'aider et t'éviter des tests en trop ?

-Ca pour le coup c'était pas si chronophage que ça. Je vais te montrer l'animation pour être calé sur le son. D'abord le code, pour voir quand le strip disparaît ou non (à l'écran).

-Tu disais justement que ce n'était pas facile de caler l'audio et vidéo ?

-Oui, parce que tu veux les caler dans le temps, donc y a pas 36 façons. Soit tu fais un truc procédural: tu génères des données, t'as une fonction, et tu génères le son et le visuel sur ces fonctions. Soit tu as les deux ressources séparées, et tu dois trouver les temps et composer les temps au bon moment, soit en général t'as un fichier son, tu l'ouvres par exemple avec Audacity, et je regarde à quelle milliseconde "radar radar" se passe, je mets des petits marqueurs et regarde l'échelle du temps. C'est un peu à l'arrache mais ça passe. Et donc je dis que de 0 à 300, l'opacité augmente. Je regarde les temps et les notes sur un bout de papier et ça c'est chiant en vrai. Ce qui est chiant, c'est que tu ne peux rien transformer. Si tu modifies ton fichier son, faut tout changer dans le code. Dans Smala, c'est pas bien outillé je trouve, dans d'autres langages, tu peux créer des points dans le temps et des valeurs et il interpole pour toi. Là tu dois faire les maths toi-même, définir comment ça marche les fonctions. Je calcule un pourcentage de cette région à chaque fois.

-Et là justement, tu imaginerais quoi pour t'aider ?

-Quelque chose pour interpoler plus facilement, des timelines, y en a dans plein de langages, dans Qt. Ou travailler avec des images-clés comme dans Animate. Tu définis des points dans le temps et il interpole pour toi ou avec des fonctions que tu décides. Mais tu dois toi quand même aller créer le lien. Donc ici j'utilise un switch range dans une FSM (le range définit les points du temps, définis à partir du fichier audio). Mais j'aimerais raisonner en graphique. Ce qui est très difficile, c'est de faire plusieurs animations et de les synchroniser. Mettons quand tu cliques là pour fermer ta fenêtre, tu veux un son et une anim' qui changent l'opacité et que tout se déplace vers le bas. Tu veux que tout se fasse en même temps, mais peut-être que ça va se faire un peu trop vite, peut-être que tu veux que ce soit plus long. Donc t'es obligé de créer plein de variables, c'est pas forcément hyper fastoche de transformer ça. L'autre approche que j'avais eu dans PyCharm…voilà attends je te fais la démo. Ca c'est codé avec une modulation, j'ai une fonction qui génère un changement, ici un sinus, et un compteur que je fais évoluer. Et j'applique cette fonction à l'amplitude sonore et à l'opacité.

-Tu aurais un exemple en Smala où tu as choisi d'appliquer une fonction de la même façon pour synchroniser plusieurs animations ?

-Ici (voir code cookbook audio), donc j'ai un sample, et on a une librairie qui nous permet de modifier des données, du son. Donc en

gros je vais faire une animation, avec une clock, donc c'est une fonction. Ma modulation c'est un sinus, et en fonction du temps où je suis et de la modulation, ça va me donner une sortie et ce truc là (abs.) "valeur absolue" me permet de contrôler le filtre passe-bas. Ce qui est délicat, c'est de faire les mappings, ie. quelle fonction de transfert tu mets entre la sortie de ta modulation et tes paramètres. Moi, ce que j'aimerais trop, c'est de lancer cette appli-là, tu testes, et tu viens dans ton code et tu ajustes et ça s'ajuste sur l'appli.

-Tu peux revenir sur ce que tu évoquais tout à l'heure, à savoir que c'est difficile de coder plusieurs animations qui doivent se synchroniser en même temps? Dis-moi si je me trompe, mais tu semblais dire que la chose embêtante, c'est que si tu veux réajuster un paramètre pour une animation, il faut souvent réajuster manuellement les paramètres des autres animations pour les adapter ?

-Oui, si t'as pas bien créé des variables, ça devient compliqué. Il faut en créer plein, alors que t'as pas envie de le faire à la mano. Je peux te montrer un exemple sur des fichiers son. En gros l'idée, c'est que les musiciens veulent régler plein de flux de données qui doivent marcher ensemble, et avec la musique faut être synchro. Donc on avait un modèle de calcul, et ce qui était intéressant, c'est qu'on savait créer des points master/pivots qui pouvaient faire tout bouger autour d'eux. On faisait de la remontée de pivot, qui servent de point de synchronisation.

-Et c'est à quelque chose comme ça que tu penses pour aider à coder les animations ?

-Oui voilà, et surtout ce qui est sympa c'est de pouvoir faire des pivots temporels indiquant le début et la fin, et qui vont contrôler et synchroniser plusieurs trajectoires de son qui sont sur la timeline, ils contrôlent tout le monde, ils translatent toutes les sources. On obtient des objets de plus haut niveau.

-Et ça existe comme structure disponible dans des langages ?

-Dans des langages, j'ai jamais trouvé ce formalisme, en tout cas pas pour ce genre de transformation. Que dans des logiciels de son.

-Tu aurais un exemple d'animation complexe où tu aurais beaucoup d'animations synchronisées ?

-Bah en fait j'évite, j'en ai pas trop pour l'instant. Je sais pas mal faire audio et son, avec cette technique que je t'ai montré dans Smala, et je fais des choses assez similaires dans d'autres langages

-Tu pourrais imaginer quelque chose pour ça, une structure de langage ?

-Bah y a déjà les images-clés dans les logiciels d'animation comme Animate, mais ça te force à raisonner avec des images clés et en seconde, y a rien qui est flexible en fait. On peut modifier les choses que sur les images clés et ensuite manuellement interpoler. Ce qui est chiant, c'est qu'on aimerait travailler les éléments indépendamment et de ne pas toujours être sur la timeline. J'aimerais

avoir des petits modules que je peux concevoir chacun. Là tu fais tout en même temps. Tu ne peux pas faire des choses indépendantes et ensuite les bidouiller.

-Est-ce que récemment tu aurais un exemple d'animation qui ne fonctionnait pas comme tu voulais et que tu aurais mis du temps à comprendre ?

-C'est souvent lié au toolkit. Genre changer le volume ça ne marche pas et il faut passer par autre chose. Ce qui est chiant après, où je dois m'y reprendre à cinquante fois, c'est toujours la même chose, c'est les courbes de modulation, comment tu veux faire osciller, varier. Y a tellement de trucs pour faire une forme d'onde qui sont trop classe, que vraiment c'est pénible de le faire avec des maths quoi. Moi tu vois j'ai un mode sinus, j'utilise le sinus, je fais le compte fois 2 pi, ensuite x 0,5 divisé par 2 pour l'aplatir. Mais après si tu veux des fonctions log c'est chiant. Ce que je voudrais, c'est pouvoir manipuler les courbes. D'avoir un lien entre le code et la courbe. Je veux un petit repère qui me donne les valeurs min max et une unité de temps et moi là, je lui dis que par défaut un sinus c'est comme ça, mais que je veux le remonter, l'écraser un peu, pour avoir ce que je veux. Mais après ce que j'aimerais faire, et je sais pas le calculer, c'est d'avoir un point de contrôler pour dire à partir d'où monter plus vite ou moins vite. Je pourrais faire des variables, mais c'est chiant quand tu développes, parce que rien ne devient modulaire. Alors qu'un éditeur de courbes et d'anim' qui sortent un truc, ce serait utile. Au moins il y aurait une sémantique beaucoup plus facile, ça ferait pas la synchronisation mais ça rendrait le truc beaucoup plus modulaire. Un truc où tu pourrais déclarer des pivots et les faire matcher avec des anims. Enfin un moyen d'exprimer que ça ça doit matcher ça. Ou dans l'autre sens: caler les anims sur les pivots déclarés dans le son. On pourrait d'un coup monter dans la hiérarchie en définissant des petits modules.

-Dans tes projets Smala récents que tu montres à l'écran, tu penses à autre chose ?

-Je n'étais pas habituée aux FSM. Maintenant que j'ai appris à m'en servir ça va, j'aime bien ce formalisme. Les FSM imbriquées ça me dérange pas trop. Après il y a des trucs que j'aimerais bien abstraire un peu. Quand c'est juste à deux états, c'est très facile. Quand il commence à y en avoir plus, c'est un peu chiant.

-Tu as un exemple ?

-Alors je peux te montrer le soundmanager. Et voilà la liste des transitions et c'est pas chouette.

-Et qu'est-ce que tu aimerais avoir là plutôt ?

-Bah par exemple _reset, tous quand ils ont _reset, tous les états vont vers idle. Donc moi j'aimerais dire "state machine go vers idle, quand y a _reset". J'aimerais factoriser ça: pour tout le monde (éventuellement sauf lui). Genre fsm.allstates. Mais ça, ça va avec l'idée que j'aimerais bien en général dans Smala, c'est les listes. Et ici par exemple pouvoir dire fsm.list.states, l'équivalent d'un "for each", tu vas vers idle quand y a _reset. Parfois en plus tu as

des états vides, juste pour stocker des trucs, ça rajoute beaucoup d'états.

-Là ça t'aiderait de pouvoir la visualiser ?

-Oui. Tu as déjà vu tourner SwingStates ? Ca c'est pas mal, moi je l'ai utilisé à un moment quand je codais. Pour voir l'état courant et voir que c'est le bon. Ca, ça m'aiderait aussi. Et aussi que ce soit coloré. Que tout ce qui va vers radarstate ce soit coloré par exemple. Et ce qui serait sympa aussi, c'est de les réorganiser dynamiquement. Fais-moi voir tout ce qui vont vers radarstate. Qui peut m'amener dans cet état. Et parfois tu veux plutôt savoir qui m'emmène là. Pouvoir réorganiser pour pouvoir se rendre compte de ces trucs. Quand tu réfléchis, quand tu la créés, tu penses à de là je veux aller là, de là je veux aller là. Mais après quand tu debug…tu te dis qu'est-ce qui s'est passé, pourquoi je suis ici? Et donc quand tu vois ça, tu te demandes quel événement propage ça, est-ce que je suis bien cohérent de partout? Tu prends un checkstate par exemple, déjà pour le voir, ou les mettre à côté, d'autres choses que des vues de FSM qui existent déjà.

I6

12/01/21

-Alors sur le plan de la programmations classique, avec relativement peu d'animations dans l'ensemble, j'ai fait essentiellement du Java, et dans Java, je fais essentiellement du JavaFx. Sur la programmation d'application, mon expérience, c'est principalement en Java. Sur mon poste précédent, où ça commençait à faire gros (mais j'étais seul, donc ce n'était pas non plus un gros projet logiciel), je faisais du Flex. Flex dans le framework Flash. Donc il y avait la partie Flash que tu faisais en programmation visuelle comme dans Animate (ça permettait de faire du dessin "animé"). Mais Flash lui-même ne proposait pas des widgets. Donc au fur et à mesure, ils ont rajouté un autre framework qui vient s'exécuter sur la même machine virtuelle, qui s'appelle Flex, dans une série de deux langage qu'ils ont créé eux, ActionScript, qui ressemble beaucoup à Java dans sa dernière version et FXML qui est un langage de description de graphe scène à balises, c'est du XML avec son propre langage dans les balises, il permet aussi de décrire des interactions. Et tu peux mélanger à volonté ActionScript et FXML. Et c'est un truc que tout le monde a pompé. Qt à la base c'est une librairie pour C++, donc c'est du code, mais à côté tu as du QML et c'est la même chose, c'est basé là sur du JSON, tu décris ta scène, tu peux rajouter des interactions pour les petites

applications. JavaFx a fait pareil aussi, les librairies c'est du Java et c'est du FXML pour la partie balises.
Le projet dans le cadre de CESAR, c'était l'évaluation et la visualisation de la complexité dans le trafic en route. Et la complexité, c'est tout un thème. Donc avant tout c'était de l'optimisation avec des réseaux de neurones. Le parti pris: on n'évalue pas la complexité car c'est trop dur à définir dans le cadre du contrôle aérien, mais de passer ça dans une boîte noire qui évalue plutôt les probabilités de grouper ou de dégrouper les secteurs de contrôle. Moi j'étais arrivé au projet à ce stade. Le but c'était de prototyper un outil. Le projet n'a pas fini mais j'ai passé trois ans et demi à la DTI R&D. Financé par CESAR, on travaillait avec X.

-Dans ton cours sur les animations, vous utilisez Animate ?

-Oui, c'est la suite Creative Cloud, c'est le descendant de Flash pro et donc tu fais de la programmation visuelle, tu fais du dessin et à côté t'as un système de timeline, tu fais des images-clés, et du choix d'interpolation et entre les deux c'est la machine qui calcule.

-J'ai beaucoup de questions sur ton cours sur les animations, mais on peut peut-être commencé par parler du projet CESAR ?

-Alors le code ne tourne plus parce que Flash est mort, mais je peux te montrer.

-Tu aurais des souvenirs de choses qui avaient été difficiles à coder ?

-Alors ouais, justement. Cette articulation (et c'est la même chose dont on parlait et qu'on peut rediscuter sur mon cours de design des animations) entre code et animation. C'est déjà compliqué de concevoir des animations. Et ça l'est encore si tu dois la faire dans ta tête et la coder. C'est pour ça qu'on cherche une programmation la plus visuelle possible. Des trucs qui ont trait au design graphique, si en plus y a un aspect temporel, qu'il est animé, c'est très compliqué à concevoir et coder. Et justement dans ce projet-là, j'avais vu la puissance que pouvait avoir Flash, alors pas seulement la puissance d'expression, mais l'accessibilité, la facilité de concevoir et réaliser des animations en faisant du dessin et en bidouillant après avec les images-clés dans les timelines. En revanche, quand tu veux faire des gros systèmes, tu es obligé de faire des trucs avec du code. Et donc Flex, c'est du code. Et même dans le cas du même framework Flex où tu peux s'exécuter en même temps, tu as quand même des barrières. Au final, j'ai tout fait en code. J'ai pas pu faire de Flash. Ca c'est pour moi la grosse difficulté.

-Et ce qui est difficile, c'est donc de faire le lien entre les deux ?

-Oui, c'est la même machine virtuelle qui exécute, le FlashPlayer, mais c'est quand même deux systèmes.

-Et malgré le fait que Flash te génère automatiquement du code, c'est difficile à intégrer ?

-Oui, je peux te montrer. Dans Flash, ça génère des machins que tu ne peux pas complètement considérer comme des widgets. Ce code généré n'est pas utilisable tel quel dans une application plus grosse. Y a pas mal de manips' à faire. Bon après, il faut aussi reconnaître que y a quand même une séparation un peu naturelle : la partie que tu programmes visuellement, elle est quand même très "cartoonesque". Et moi ce que j'avais à faire dans le projet, c'était des trucs très dynamiques et analytiques, sur la base des résultats des algos. Donc il fallait mettre du code de toute façon pour intégrer ces résultats et générer des scènes dynamiques. Donc c'est aussi une différence qualitative.

-Et tu retrouves le même problème avec Animate ?

-Oui, parce qu'Animate a pris le même framework importé côté Flash. Et il n'y a rien de prévu pour faire des trucs plutôt WIMP. Et le fait est que par ailleurs, il y a des tas et des milliers de librairies en JavaScript. Donc il y a toujours une collision quand tu veux utiliser plusieurs technos pour des buts un peu différents, mais dans le cadre de la même appli'. Mais je ne sais pas si on peut trouver une solution à ça, parce que c'est un problème d'intégration. Et tu l'as par ailleurs, au delà de la programmation visuelle, dès que tu veux utiliser une autre techno. C'est un problème commun à tous les systèmes.

-Et si on laisse de côté le problème général de l'intégration, est-ce qu'il y a quelques difficultés spécifiques qu'on peut isoler, des choses très identifiables, dans l'intégration de la programmation visuelle à du code textuelle? Notamment avec Animate, quand tu veux intégrer ton animation à un programme Javascript?

-Il faudrait être capable de produire un truc, quel que soit le choix que tu fais, ou code ou programmation visuelle, inter-opérable, quel que soit l'environnement d'exécution. Qu'on puisse utiliser les deux indifféremment. Mais ça c'est pas gagné.

-Quand tu définis dans Animate des temps sur ta timeline, est-ce que c'est facile d'y faire référence dans ton code ensuite ?
-Alors oui c'est facile. Alors typiquement t'as un numéro d'image auquel tu peux faire référence, mais en plus t'as ce qu'on appelle des étiquettes ou des labels d'image, là c'est plutôt pour les images-clés, donc t'as un nom clair, sans problème d'ambiguïté de nom, sans te tromper d'indice. Donc tu peux dire "dans le cadre du jeu cette animation, rends-toi à telle étiquette et joue l'animation, ou arrête-là". Et donc t'as des instructions simples depuis le code.

-Et tu peux renvoyer tout aussi bien à des temps dans l'image ?

-Alors non, il n'y a pas de temps. Il y a que la notion d'image. Si tu veux un temps, il faut calculer. Alors ça ce n'est pas forcément bien par contre. Si tu veux un temps, il faut calculer l'image qui correspond en fonction du *frame rate* de l'animation. Toutes ces données sont accessibles mais il faut le calculer.

-Et parfois c'est une limite justement, de devoir calculer? Est-ce qu'on aimerait parfois raisonner plutôt en "temps" plutôt qu'en "image" ?

-Oui. Mais alors s'il faut choisir l'un des deux, je pense qu'il faut rester sur l'image, parce que tu définis un frame rate, des images dans ton dessin, et l'image pour moi c'est la partie la plus bas niveau. Donc si on doit faire qu'une chose, c'est naturel que ce soit l'image. Par contre, tu devrais pouvoir automatiquement via l'API avoir le choix de travailler sur les images ou le temps, voire une portion de l'animation. "Je veux me rendre à la moitié de l'animation". Autant que l'API le propose, même si les calculs ne sont pas compliqués. Je peux te monter les exemples sur les travaux des étudiants. C'est un petit jeu de voitures avec des interactions. Et ils sont évalués sur les animations qu'ils ont faites (démo). Donc le principe, c'est d'avoir des timelines dans chaque composant, une timeline pour la scène globale. Typiquement pour un jeu, tu vas pas faire jouer les choses là-dedans. Tu as deux images-clés (…) un symbole — c'est une classe pour un codeur. T'as rien en WIMP, enfin tu peux créer des boutons mais c'est tout. La représentation, elle est super adaptée. Tu fais des dessins, puis tu fais tes images-clés et entre les deux des interpolations.

-Les élèves ont des guidelines? Par exemple, je veux créer l'impression qu'un objet tombe vite? Genre pour tel effet physique, utiliser tels paramètres et telles valeurs ?

-Alors oui, je donne des conseils, mais c'est les miens. Je pense qu'on peut les trouver quelque part mais je n'ai pas fait l'état de l'art. T'as plein de trucs pour ça dans l'appli'. On est dans un domaine à la croisée des disciplines, entre le graphisme et les codeurs. Les codeurs ne font pas ce genre de choses. Et en revanche, ces trucs c'est plutôt pour les graphistes, et eux ils ne font pas d'état de l'art, ils ont eu des cours où on leur dit de faire comme ci ou comme ça.

-Donc il y a peut-être une zone un peu grise entre les deux ?

-Oui, c'est intéressant, mais j'ai pas eu le temps de creuser. Après typiquement, par exemple regarde l'exemple du métronome que je leur donne à faire. Ici, c'est une animation qui doit se jouer en boucle. Le principe de base, c'est que ta première et ta dernière image-clé ça doit être les mêmes. Alors en toute rigueur c'est pas tout à fait ça. Donc voilà, on a des petits conseils comme ça. En fait, c'est de la création artistique, et je ne sais pas si beaucoup de gens s'y sont intéressés. Et s'il y en a, ça ne m'étonnerait pas qu'on ne les utilise pas, et qu'il n'y ait pas eu de pont entre le monde du design et les codeurs. Tu apprends ça de façon plus ou moins explicite plutôt dans les écoles de design graphique. Par contre, clairement ça a été pensé (dans Animate). Regarde. Typiquement sur un métronome, sur le rendu, la vitesse est relativement réaliste. Ou tiens (autre exemple démo, travail d'étudiant). Tu vois que le pendule n'est pas réaliste ici. Ce qu'il y a dans notre domaine, c'est qu'on peut vouloir des guidelines, mais en fait c'est facile de les trouver seul, on a fait des études scientifiques, on sait pourquoi c'est réaliste ou non. Ici ce pendule n'est pas réaliste. Sur un pendule, la vitesse n'est pas uniforme, tu ralentis, parce que t'as ton énergie potentielle qui augmente. Donc ici il devrait avoir une accélération et ralentissement. Et même si tu ne connais pas la physique, tu comprends. Et si tu sais ce qu'il faut faire, c'est facile de le réajuste (*dans*

Animate). Pour cela, tu rajoutes une accélération entre tes deux images-clés. Tu choisis deux effets ease in et out des deux côtés.

-Donc ce n'est pas une nécessité d'avoir des guidelines ?

-Oui il y en a besoin mais c'est pas des trucs de haute volée scientifique. C'est de l'observation. Mais c'est vrai qu'il y en a besoin. Quand tu enseignes, il faut bien donner des règles aux élèves, sauf à faire le pari qu'ils vont trouver tout seul.

-Y a des choses qui seraient frustrantes, en plus du besoin de vouloir raisonner avec du temps parfois ?

-Oui, alors des tas de choses. Mais c'est lié à des principes hérités de FlashPro qui a vingt ans. Donc il y a des trucs qui ont été construits incrémentalement, et qu'ils n'ont jamais remis en cause, de peur de dérouter les anciens utilisateurs. Ou de peur de devoir tout redévelopper. Donc typiquement, il y a des trucs vraiment pourris du genre, quand tu veux travailler sur les dégradés. Bah il y a quatre endroits pour le faire, c'est pas unifié. Donc ici par exemple, tu créé ton ellipse, ensuite tu dois la remplir et pour ça il faut aller dans la palette de couleurs, choisir que c'est un dégradé radial, et ensuite pour régler finement ton machin (et d'ailleurs c'est pas top non plus avec Illustrator), tu dois accéder à un autre outil, la transformation de dégradé, et là tu choisis, tu déplaces le point central. Donc c'est des trucs qui manquent d'unité, qui ont été rajoutés petit-à-petit.

-Il y aurait d'autres choses, et notamment ensuite sur l'intégration de l'animation au code dont tu parlais ?

-Alors ouais, ça génère du code et c'est un fichier JavaScript, particulièrement illisible, même dans l'organisation du graphe de scène c'est compliqué. Clairement, l'esprit d'Animate, ça n'a pas été de créer des composants réutilisables ailleurs dans d'autres technos. Alors qu'on est en plein dans ce qu'il aurait fallu faire pour du techno web. Normalement, on devrait pouvoir générer un bout d'animation, un widget, que tu mets n'importe où dans la page web. Eux, pour ça, ils génèrent une page HTML – pas énorme, mais quand même pas mal de merdes. Plus un fichier JavaScript, qui lui est énorme, décrit l'ensemble de la scène, tout ce que t'as fait dans l'animation. C'est tellement merdique, qu'on ne regarde même plus le code (généré). Si tu veux utiliser ce que t'as fait dans Animate tel quel, c'est super facile. Et donc si tu veux faire autre chose, il faut comprendre le code généré et l'intégrer dans ton application web, et ça, il n'y a rien qui permet de le faire. Faut le faire à la main.

-Et tu disais que faire référence à des éléments de ton animation dans ton code, c'est pas compliqué par contre? Je veux dire changer un paramètre de ton animation depuis le code ?

-Alors c'est assez compliqué, il n'y aucun moyen documenté de le faire. Par contre c'est possible, je suis allé le hacker le truc. Mais clairement ça n'a pas été prévu, mais je ne peux pas imaginé qu'ils ne l'aient pas envisagé; à mon avis, c'est qu'ils veulent pas, ils veulent qu'on continue à utiliser ça en spécialistes de l'animation, sans faire le bridge entre les deux. Ici (fichier JavaScript généré)

dans ce bordel, t'as une variable globale qui est cachée, et je n'ai jamais réussi à trouver dans la doc d'Animate une mention de cette variable. Mais cette variable permet d'accéder à la scène. Et ça je peux te montrer (démo), c'est une animation avec un compteur. J'ai généré un fichier HTML qui intègre mon compteur (en Animate), et je commande par le code de la page l'animation. Mais pour ça j'ai dû hacker le truc. J'ai repris la page générée et j'ai rajouté mes scripts à moi en faisant une nouvelle page. Je dois recréer des variables. Alors que ça coûterait rien de permettre d'exporter les éléments comme des composants que tu peux exploiter. Qu'on ait juste à faire référence au fichier JavaScript. C'est d'autant plus bizarre, que le code du JavaScript, c'est pas une techno propriétaire. C'est le gros truc que je reproche, de pouvoir faire la navette quoi.

-J'ai une question plus précise, quand tu écris une animation en code, que ça compile mais que tu n'obtiens pas le résultat visuel que tu veux, tu as une stratégie pour réajuster ou comprendre le bug ?

-Alors en code c'est différent. Ici avec Animate, t'as pas ce problème justement. Le résultat visuel tu l'as en temps réel, tu navigues à la main, tu vois bien que ça ne fait pas ce que tu veux, ça ne t'explique pas pourquoi mais vu que tu vois le résultat visuel…alors que quand tu codes…alors là c'est la merde. C'est déjà compliqué avec le code de base, c'est encore plus compliqué avec du code graphique, parce que tu ne comprends pas toujours ce qui est affiché. Et alors quand tu as en plus l'aspect temporel qui se rajoute à ça…

-Et qu'est-ce qui fait qu'on ne comprend pas en fait ?

-Du code arithmétique, c'est compliqué, mais tu peux accéder aux états intermédiaires par des points d'arrêt et voir le contenu des variables. Et ce que tu as conçu dans ton algorithme pour faire des calculs, tu en conçois les étapes intermédiaires. Avec du dessin, si le rendu c'est n'importe quoi, t'as aucune indication pour comprendre pourquoi c'est n'importe quoi. Donc du coup tu dois faire la même chose, tu dois revenir à l'étape d'avant comme avec les calculs, sauf que dans l'espace graphique, c'est pas juste le résultat d'un ou deux calculs. Y a plein de dimensions, c'est pas juste le résultat d'un calcul. Bon après, je caricature peut-être, parce que j'ai pas l'habitude de faire de l'optimisation. C'est lequel de tous ces chiffres qui merdent ? Mais dans le dessin, c'est systématiquement comme ça. Ne serait-ce qu'un pixel, c'est déjà deux à trois coordonnées, c'est trois à quatre chiffres pour coder la couleur et la transparence et c'est des relations avec d'autres pixels. Donc je pense que la complexité explose à cause de ça essentiellement. Et dans l'aspect temporel, c'est la même chose en temps réel qui évolue. Donc il y a encore une magnitude. Donc t'as la situation que tu n'arrives pas à debugger, sauf qu'elle arrive 24 fois par seconde.

-Donc tu dirais qu'il y a deux problèmes, complexité et temporalité ?

-Ouais, les deux étant liés, la temporalité fait exploser la complexité, de la même manière que le côté graphique rajoute beaucoup de chiffres à comprendre et vérifier. Avec la temporalité, il y a une difficulté conceptuelle de plus.

-Si on met de côté Animate, pour du code textuel tu as déjà imaginé un outil qui aurait pu t'aider pour des animations?

-Non, jamais pris le temps de réfléchir à une manière de représenter ça. Effectivement voilà, avec Animate c'est facile, on fait du dessin figuratif, c'est pas du dessin calculatoire, donc c'est plus facile à debugger. Tu vois que le bras (référence à une des démos-animation précédente) il n'est pas à l'endroit que tu voulais. Tu sais que ton image-clé de départ ou de destination n'est pas comme tu voulais, ou que ton interpolation n'est pas la bonne. Si c'est du dessin calculatoire, tu te retrouves dans le même cas que tout à l'heure: la complexité a explosé et toi t'as aucune indication visuelle. La timeline rend intuitive la représentation dans le temps, et c'est essentiel quand tu veux penser du data flow.

-Et les interpolations qu'ils proposent, elles sont toujours satisfaisantes?

-Non. Alors y a trois types d'interpolation. C'est un des trucs les plus riches. Il y en a deux qu'on peut considérer comme identiques. Il y en a une c'est l'interpolation traditionnelle, qui travaille sur des occurrences (en *Animate*, un symbole, c'est une classe, c'est une recette pour construire un objet, et l'occurence c'est l'instance de la classe). Donc le principe, c'est que tu définis des symboles, et dans l'interpolation classique, tu travailles sur un symbole, un seul. Et sur ce symbole, le même dans les deux images-clés qui entourent l'interpolation, tu fais subir un certain nombre de transformations, mais ces transformations sont répercutées sur toutes les autres occurrences dans la scène et d'autres non, et la difficulté c'est de savoir lesquelles se répercutent sur le symbole/la classe. Après, tu réalises vite si tu te plantes. Donc pour faire l'interpolation classique, il faut que je change des propriétés qui ne s'appliquent que sur l'occurence. Alors typiquement, c'est avec l'outil de transformation libre, et je peux par exemple changer l'échelle, faire des rotations, cisailler, des transformations géométriques, je peux appliquer des effets colorimétriques. Par contre, je ne peux pas changer la couleur, la couleur c'est une propriété intrinsèque aux symboles, je change toutes les occurrences. Et pour moi c'est pourri ça, et c'est un héritage de l'ancienne construction de Flash je pense, enfin en tout cas je ne vois pas d'avantage à ça. On préfèrerait avoir l'approche du codeur: si tu modifies l'objet, une instance, ça ne modifie que lui. Donc voilà la difficulté dans l'interpolation, c'est de gérer si elle s'applique sur une occurrence ou sur tout le monde. T'as pas ce problème en JavaFx ou Smala. Donc la deuxième méthode, c'est l'interpolation de mouvement, elle fait la même chose mais en mieux mais en plus coûteux pour les performances. Grosso modo, c'est une interpolation qui travaille sur un set de propriétés de l'objet. La troisième, est finalement assez peu courante, c'est l'interpolation de formes, et ça c'est une boîte noire, tu ne sais pas du tout comment ça transforme. Ca peut être transformer un carré vert en un carré rouge, et elle saura faire le morphing de couleur, elle saura travailler sur l'opacité. On peut animer un tas d'autres choses, mais ça ne travaille que sur une forme et c'est une boîte noire. Donc sur des trucs simples, ça donne exactement ce que tu penses que ça va donner, sur des trucs compliqués ça donne juste des trucs pourris, et c'est tu sais pas, tu dois multiplier les essais, tu dois séparer ta forme en plusieurs sous-formes pour faire plusieurs interpolations,

si tu penses pouvoir isoler des choses plus simples à faire. Et t'as un truc qui serait vachement bien, c'est les repères de formes, et ça fait depuis quelques versions que je ne peux pas l'utiliser.

I7

15/02/2021

-Je bosse depuis 2006, donc beaucoup j'ai eu beaucoup de projets et de types différents. Il y a toujours une partie IHM. C'était ppliqué au contrôle aérien en début de carrière, par exemple mon projet de recherche Erasmus. Et ensuite, j'ai fait de la recherche en IHM chez I., notamment une IHM pour faciliter la communication entre agents, destinés à divers domaines d'application dont l'ergonomie. Un thème du projet, c'était la multimodalité dans les cockpits. J'ai fait plusieurs projets tout seul, souvent à deux, et le max: quatre ou cinq développeurs.

-On peut commencer par regarder tes projets les plus récents ?

-Ce qui est peut-être le plus facile pour moi, c'est l'éditeur. Ca va devenir un produit de la société I. Un peu de contexte peut-être : l'idée de ce logiciel, c'est de faciliter les échanges de communications entre agents. Un agent, c'est une boîte, et tout le monde discute sur un même bus logiciel. C'est un peu comme Ivy si tu veux. Ici, on voit la liste des applications qui apparaissent, avec une définition — toutes les entrées, toutes les sorties. Et dans l'idée, ça c'est un agent qui produit du son par exemple, bah si je connecte mon entrée sur cette sortie, dès que cet agent là va écrire sur sa sortie, lui son entrée va être automatiquement notifiée. Tu ne connais pas par contre le fonctionnement interne d'un agent. *(Demo de l'IHM)*. Ca permet de faire du rejeu, de la mise au point au cours du développement, et tu as une notion de timeline où tu vas pouvoir positionner des actions, et quand tu appuies sur "play" ça va te jouer telle action (écrire x sur telle entrée etc.)

-Et c'est destiné à qui ?

Ca n'a pas de domaine précis d'application. C'est destiné, soit à des développeurs pour la mise au point ou à des responsables techniques dans des sociétés plus grosses, pour faciliter de la simulation ou du rejeu. Et notre cible, c'est plutôt des ergonomes. Là, on a fait une expé pour la société A.: tester différents scénarios, tester des alertes pour les pilotes (différents scénarios en jouant sur les

vibrations du siège, des LEDS, et retour physiologique comme la mesure du rythme cardiaque etc).

-Et tu te souviens de choses qui n'auraient pas été évidentes à concevoir dans le code pour ce système?

-Alors derrière cette IHM, je me base sur une librairie C développée par mes collègues. Donc au début je n'utilisais que quelques événements, puis ensuite tout ce que permettait la librairie. Donc un agent apparaît sur le réseau, puis disparaît, une information est écrite sur une sortie. Alors pour être tout à fait honnête, le fait de pouvoir poser des événements sur la timeline, je me suis basée sur la timeline d'un autre projet, je l'ai adaptée. Mais donc la complexité, toute la partie mathématique, la logique de positionnement, notamment sur le (de)zoom, ce n'est pas moi.

-Qu'est-ce qui t'a pris le plus de temps à faire ?

-Des algos assez complexes. Par exemple, un même agent, tu peux le lancer plusieurs fois. Et donc il y avait une logique. Par exemple, là c'est un peu grisé, si jamais on lance l'agent *text window* sur le réseau, il va être détecté par l'application automatiquement, et on va changer la couleur, on va faire du feedback graphique. Et si j'en lance un deuxième, je vais avoir un petit indicateur qui me dit "j'en ai deux sur le réseau", et en plus en ayant la logique de, quand-ils-apparaissent-et-disparaissent, on a un algorithme qui dit "si je n'en avais plus aucun sur le réseau, or j'en ai un qui arrive, on va considérer que celui-là remplace les autres. Par contre, si j'en ai plusieurs d'actifs sur le réseau, là c'est bien des agents différents." En plus, un agent peut avoir le même nom mais avoir des entrées et sorties différentes en fonction des versions du code. Donc il y a tout une logique pour dire un agent donné. C'est simple pour l'utilisateur à la fin, mais par contre derrière il y a tout un algorithme pour gérer l'état (actif/inactif), présent sur le réseau ou pas, voir si les entrées et sorties sont dans tous les agents ou pas. Ca, ça été complexe. Et faire aussi marcher la logique du "il (un agent donné) apparaît ici et il apparait ici mais"…attends je te montre sur l'interface. Donc voilà ce que je veux illustrer: là c'est une seule boîte, mais là on voit deux boîtes différentes: dans un des agents, y a un certains nombres d'entrées et sorties, et tu retrouves les mêmes avec d'autres supplémentaires dans l'autre agent. Donc on parle d'un même agent, mais pas avec les mêmes entrées et sorties, mais ils ont les mêmes noms. Donc l'utilisateur va avoir besoin de vérifier si c'est un problème, si c'est un problème de version etc. Donc il y a des feedbacks qui mettent en alerte l'utilisateur. Ca c'était compliqué.

-Tu peux m'expliquer en quoi exactement ?

-C'était les algos derrière. Mais y avait quand même le principe de remontée d'événements. Quand cet agent-là arrive sur le réseau, c'est comme si j'avais branché un connecteur et hop il y a quelque chose qui me notifie. Donc j'avais souvent besoin de point d'arrêt pour vérifier qu'un événement est bien arrivé, puis vérifier la définition, vérifier si ou non un agent apparaît en double. Alors c'est aussi l'aspect ordre dans lequel ça intervient. C'est ça qui a beaucoup d'intact: qui est arrivé avant, après. L'ordre d'arrivée des

événements et lesquels, c'est ça qui est compliqué. C'est pour ça, quand il se passait un truc qui n'était pas logique, j'utilisais un point d'arrêt, pour regarder dans quel ordre se sont passées les choses. A partir du point d'arrêt, je regarde les valeurs des fonctions appelées ou les événements émis dans le code, la stack. Pour Qt, en IDE j'utilisais Qt Creator.

-On peut revenir sur le promet d'ordre d'arrivée des événements et de quand il se produisait quelque chose d'inattendu? Tu procédais comment?

-Un point d'arrêt qui me permet de remonter et de voir. Je suis sûr que c'est passé par là, mais pourquoi? Ou à l'inverse je mets un point d'arrêt pour faire du pas-à-pas.

-Il t'est arrivé d'utiliser d'autres outils pour t'aider à debugger ?

-Non jamais, pareil pour les autres collègues, surtout les stacks, les libérations mémoire. On utilisait Qt Creator ou Xcode. Avec la timeline dans QT Creator, c'est quand même très bien foutu pour voir la création des objets. Après en général, surtout quand je découvre quelque chose, comme là avec Matsuri, je fais que des logs. D'ailleurs je peux te montrer les outils de debug que j'avais fait pendant mon stage chez I. C'était un arbre. Représenter l'arbre Djnn, avec la racine et les différents noeuds, dont la taille était proportionnelle à la descendance. C'était dynamique. J'avais aussi travaillé sur les FSM et les Rules (switch). (Demo). Ca tournait en parallèle de l'application. On pouvait afficher à la demande les *bindings* et les symboliser dans l'arbre. C'était codé en Intuikit.

-Tu veux qu'on discute de projets antérieurs ?

-Alors il y a un truc qui me vient, typiquement les FSM, c'est un des trucs principaux où je me suis dit c'est chouette que ce soit autant matérialisé (dans Intuikit ou Djnn) et que c'est pas forcément dans d'autres langages ou IDE. WPF le fait, il y a des machines à états finis, tu peux même configurer les animations et les transitions, c'est assez pratique. Mais dans Qt ils ont un truc, c'est assez pratique mais pas très utilisé. Et en Objective C, pas du tout. Ca me fait penser que je peux te montrer un autre projet significatif que j'ai fait il y a quelques années, sur la multimodalité dans les cockpits. Ca s'appelait Iode. (Demo) Je pense à ça, parce qu'on avait plein de FSM. On les traçait d'abord sur papier, avant de les coder, donc c'était vraiment dans l'esprit Djnn. On projetait sur le carton une application, on remettait les outils que les pilotes ont l'habitude d'utiliser, avec en plus de la modalité tactile avec des capteurs derrière le carton qui nous permettait d'interagir, le *leap motion* qui nous permettait de pointer en 3D partout sur le cockpit, et la dernière interaction, y avait du tracking du regard sur la roadmap, pour faire le parcours de l'avion. Donc il y avait des FSM partout et une FSM globale qui dit "j'ai détecté du leap motion, donc ça prend le pas sur la détection du regard". Des notions de priorité comme ça. Ou "si j'ai un item graphique, que je le regarde, on va avoir un feedback, et si je confirme sur la tablette tactile, on va encore passer dans un état différent." Une multitude d'états et des FSM imbriquées avec des switch.

-Et même si ça se prêtait bien aux FSM et que ça aidait, est-ce que parfois ça a pu poser des difficultés aussi ?

-C'est forcément arrivé au moment de la mise au point, "là on n'est pas dans l'état qu'on veut", "pourquoi, je suis dans quel état? Et si je suis dans tel état, pourquoi? J'ai peut-être pas bien capté l'événement au moment où je croyais, ou qu'est-ce qui m'a amené là?".Après aussi il y avait plein d'OS différents (pour la reconnaissance de voix sur une machine Windows, le pfd remote, l'*eye tracking,* le *leap motion*, deux IHM, la tablette tactile du pilote et la *roadmap*). Et donc pour faire communiquer tout ça, *n* machines et *n* OS. Donc il nous fallait faire communiquer tout ça en utilisant un même langage.

-Et justement, assurer la communication de tous ces applis, ça a représenté quoi comme partie du travail ?

-Quand même pas mal. Fallait toujours aller vérifier pourquoi on n'avait pas ce qu'on voulait sur un affichage; ça ne venait pas forcément d'un bug dans l'application, mais est-ce que telle application est bien sur le réseau, est-ce que le device est bien lancé ? On avait un peu une logique du debug avec un outil chez I. (demo) : ici si besoin, on peut aller voir l'historique des sorties, on a un *timestamp*, donc on a toutes les valeurs qui sont apparues sur toutes les sorties, comme ça tu peux revoir tout l'historique de tout ce qui a transité sur le réseau. Après en général, ce qui me fait chier, c'est l'aspect quantité de données. Tout à l'heure, tu sais, on parlait des points d'arrêt ou printer. Mais si tu as du rejeu qui te burine et tout, à un moment les logs c'est limite, mais c'est vrai aussi pour le point d'arrêt: s'il se passe trop de trucs, t'essayes de raccourcir la fenêtre qui t'intéresse, et puis si tu bloques trop longtemps, tu peux faire péter les applis, ou si tu mets un point d'arrêt, ça fausse le contexte.
Ensuite pour le problème des jeux de données, ça dépend aussi des projets. Même si ça pète et que tu es autonome et que tu as accès au gros jeu de données, ok ça pète mais tu fais chier personne, tu vas améliorer les perfos, tu vas voir ce qui n'allait pas, et ça ne sera pas gênant. Mais si tu ne peux tester que dans ton coin et après les tests, c'est solliciter trois personnes, trois ordinateurs, et faire chier cinq personnes, et que ça croûte tout le début…c'est embêtant. Après c'est tout l'intérêt de la simulation, pour simuler une montée en charge, au moins simuler la quantité.

-Et pour gérer le problème de quantité de données, quand les breakpoints ne sont plus très efficaces, tu as déjà utilisé ou imaginé des outils qui aideraient ?

-Ce qui serait top c'est de pouvoir filtrer. Bon, si ça se trouve ça existe. Mais en gros ce serait mettre un breakpoint en disant "if… je m'arrête tous les 100". Ca m'arrive de faire ça avec des logs. C'est rajouter des conditions sur le print. Mais peut-être que c'est automatisé sur certaines IDE cet ajout de condition. En fait, ça rejoint l'idée que c'est très pratique de voir quand l'événement est arrivé, et peut-être il y a des fois où tu veux mettre un breakpoint, pour arrêter au moment où ça va se passer, bloquer et avoir la main, ou alors tu veux juste le contrôler, avoir un feedback visuel par exemple. Mais on pourrait vouloir un truc intermédiaire, ça me suffit

pas de voir que ça c'est allumé, mais c'est de voir la donnée elle-même qui a transité. Avoir différents niveaux en fonction du contexte. Et ce qui est super chiant, c'est s'il y a plusieurs threads: si tu bloques quelque chose et qu'il y a des trucs qui se passent derrière Après, je repense à d'autres problèmes qui pourraient t'intéresser. Genre sur mon IHM la fenêtre en elle-même et transparente mais il y a quelque chose en fond qui a une couleur et donc ça rejoint le truc que je t'ai dit que potentiellement on pense à un debug style breakpoint etc. Mais le debug graphique, vraiment pour aller voir quels sont les éléments affichés et surtout les couches, dans quel ordre, et ça je sais que ça pourrait être utile.
Dans la même logique, il y a quelques jours, sur un autre projet P., M. me disait, pour des histoires de perfos, à la limite ils vont charger tout un tas d'avions et balises, décalés en dehors de l'écran. Donc pareil, ça rejoint la même logique, tu codes quelque chose et ça n'apparaît pas à l'écran, tu sais pas pourquoi. Ici t'aurais besoin d'avoir accès à la liste des éléments affichés, voir les différentes couches, ça permet de voir "ah il n'est pas affiché, mais parce qu'il est en dehors de l'écran ou parce qu'il est derrière quelque chose; genre tu réalises qu'en fait t'as bien ta fenêtre transparente mais un rectangle est affiché derrière". Le principe c'est juste avoir une vue de tous mes éléments affichés, et derrière ça répond à différents scénarios et différentes problématiques. Alors je peux te montrer un outil que j'utilisais avec WPF, c'est ***Snoop***, là c'est juste une capture d'écran pour un exemple bateau. Tu vas avoir l'arbre avec la fenêtre et les composants que tu peux dérouler, pour chaque item t'as les propriétés et en parallèle t'as ton application. Y a des petits mécanismes en plus, genre si tu sélectionnes, tu peux modifier la propriété et c'est réinjecté automatique sans avoir à relancer l'application. Là c'est vraiment pour répondre à la question "pourquoi mon rendu il n'est pas conforme à mon attente?".

I8

19/02/21

-J'ai beaucoup bossé sur du flux vidéos pour des systèmes destinés à des tours de contrôles remote (projet E.), j'ai fait aussi de l'image radar, de l'OS chez B. et du debug sur le hardware et de l'intégration d'Android chez N. J'ai fait aussi un passage dans une startup où on bossait sur du Linux, et tout le monde faisait de tout, de debug noyau aux tests. Pour venir aux problèmes que j'ai… C'est la reproduction qui peut être très compliquée. Parce qu'un problème de mémoire sur un programme…le programme tourne il fait un crash, une fois qu'on a le

crash, on a tous les éléments pour rattraper. Même si c'est complexe, on a des debuggers pour revenir en arrière, avec un debugger et un core on fait tout ce qu'on veut. Le problème étant plus compliqué où on fait quelque chose, et un cas non identifié, avec un bug qu'on ne peut pas reproduire, chaque fois qu'on va réessayer de tester ce cas n'arrivera pas. Chaque fois qu'on voudra le reproduire, on n'y arrivera jamais et on ne comprend pas quel événement, quelle est la suite d'événements qui fait qu'on tombe dans ce cas-là. Je réfléchis à si j'ai des cas concrets comme ça. Souvent ce qu'on essaye de faire, c'est d'avoir une représentation temporelle. On peut prendre des logs, en mettre le plus possible, sans risquer des bugs avec le debug, en espérant que ça ne change pas les événements, parce que le simple fait d'écrire un log peut changer les événements, le fait que quelque chose arrive toujours après ou toujours avant. Et du coup dès fois un log qui marche, un log qui marche pas, essayer de voir s'il y a une différence de tempo et caractériser qualitativement.

-D'autres outils employés, autres que logs et points d'arrêts ?

-Ca dépend un peu des projets. Après moi j'ai fait du debug noyau par exemple, avec des sondes, des outils regarder les registres du CPU, du assez bas niveau ou encore des analyseurs USB qui dump l'état du dialogue USB entre device et core, ce genre de chose. L'IHM n'est pas ma spécialité mais y avait des gens qui faisaient beaucoup d'IHM, notamment sur les performances y a des outils comme PyTimechart. En gros, c'est mettre quelque chose de temporel en présentant tout ce qui se passe. Ca donne tout ce qui se passe sur chaque CPU de la machine, où sont les processes, dans quel état ils sont, quand arrive une nrq, comment elle est traitée. C'est mis dans un dessin temporel. Et pour les performances en IHM, ça nous est arrivé de nous en servir (cf. Dump Présentation d'un collègue, pdf). C'était sur une application Facebook, un problème de scroll qui lagait, il s'arrêtait puis reprenait et on n'avait pas ça sur la concurrence. Le lag, on ne l'avait pas toujours, ça arrivait de temps en temps, donc la reproductibilité n'est pas terrible. Donc on a analysé le truc en se disant, pourquoi la frame ne s'est pas affichée? Avec ce genre d'outil, on voit qu'on a raté l'affichage de la frame — la frame n'était pas prête donc elle n'est pas affichée —, et on cherche pourquoi l'événement n'a pas été traité en temps et en heure. Ici, en l'occurence, c'était une histoire de cache CPU qui était trop faible. Et au scroll, vu que les images, qui se préchargent à la volée, n'étaient pas assez vite chargées, le cache était trop petit, il y a avait de l'éviction, et donc il y avait un bloquage. Ca pour voir ça, il faut effectivement avoir des outils d'analyses assez bas niveau et temporel.

-On pourrait commencer par regarder le projet E. ? Est-ce qu'il y a des choses dont vous aurez envie de parler, qui auraient été difficiles ?

-Par rapport à la problématique, j'essaye de réfléchir. Il y a une chose qui a été particulièrement compliquée, on a eu un post-doc qui voulait utiliser ces systèmes pour faire du vrai positionnement 3D. Quand on a l'avion sur la caméra, on a une position en 2D, qu'on va transformer en deux angles pour savoir à peu près où il est, et on va le projeter sur le plan du sol pour l'avoir de nouveau en 2D. En fait, on n'est jamais en 3D dans ce monde-là. Et lui ce qu'il voulait, c'est

utiliser deux caméras à un mètre d'écart et filmer du coup deux fois la même scène et avoir les angles de l'avion sur les deux caméras; de là, on a quatre dimensions et on peut en déduire la profondeur et exactement la position de l'avion en 3D. Pourquoi j'insiste, c'est qu'on positionne par rapport au sol, donc s'il n'y a pas de sol, on ne sait pas où il est. Typiquement, s'il est en l'air, on ne peut pas le positionner. C'est le plan qui nous réduit les dimensions et nous permet de trouver la profondeur. Mais dès qu'on enlève ce plan, on peut pas. L'idée du coup, c'était de faire comme l'oeil humain. On voit les choses sous deux angles différents et on en déduit la profondeur. Pourquoi je vous parle de ça, c'est parce qu'en fait il y avait un truc assez fin, c'est que pour faire ça, autant pour un objet qui est fixe on peut filmer les deux flux, on s'en moque, autant sur un objet qui se déplace, en regardant les deux caméras, il faut que les images qu'on analyse soient exactement au même moment. On ne peut pas avoir un décalage sur les flux, faut pas que les images soient à des temps différents, parce que l'objet s'étant déplacé, cette notion de différence d'angle n'a plus aucun sens. Du coup, j'avais fait des trucs déjà pour récupérer un temps absolu sur les caméras, pour timer toutes les frames, et ensuite faire un système pour être capable, pour chaque frame de la caméra droite, d'avoir la frame équivalente de la caméra gauche, en tout cas la plus proche au niveau temps. En sachant que l'encodage des caméras, c'est une frame complète et les images qui suivent c'est un delta de l'image précédente, en gros on a une dizaine d'images comme ça. Donc on a une image complète, puis des deltas, des deltas, des deltas, une image complète etc. En vrai ce n'est pas tout à fait vrai, mais dans l'idée c'est ça. Ce qui fait qu'une caméra encodée, peut avoir des deltas très petits, c'est le cas sur un petit aérodrome où il ne se passe pas grand chose. Donc pour le décodage, on reçoit un gros paquet de frames, puis plus rien, puis un gros paquet de frames, puis plus rien, ce qui fait que quand on regarde les deux caméras, les deux trains de frames n'ont aucune raison d'être synchronisés. Quand on les lit bêtement à la volée, ce qui va se passer, c'est qu'on va voir plein de frames de la caméra 1, hop plus rien, plus aucune frame des deux côtés, plein de frames de caméra 2 et on peut en avoir une dizaine, puis plus rien. Donc faut bufferiser tout ça des deux côtés pour arriver à retrouver une sorte de synchro. Tant que la queue d'une caméra 1 n'a rien, on attend, puis elle se remplit et tant que la caméra 2 n'a pas reçu de frame plus récente, on attend. Si on est en retard, on bypass la frame 1 parce qu'on sait qu'on ne trouvera jamais côté caméra 2 une frame équivalente.

-Des difficultés particulières pour l'implémenter ?

-Au final, c'était pas beaucoup de lignes, mais c'était la logique, l'algo. Qu'est-ce que j'attends, qu'est-ce que j'attends pas et qu'est-ce que ça veut dire avoir la frame la plus près par rapport à un temps? Ce qui m'a posé le plus problème, c'est qu'à la base, il faut une sorte de temps absolu. Certaines caméras font du RTSP et c'est un protocole avec une notion de qualité de service, notamment y a tout un truc pour rendre disponible un t0 absolu de quand le flux a démarré. Et ensuite chaque frame a un timestamp. J'en reviens à mes caméras 1 et 2. Donc les deux images sont vues avec des angles différents et je veux un panorama qui soit joli et pas juste deux images côte-à-côte quoi. Donc je reconstruis une image complète comme s'il n'y avait pas deux caméras. Au-delà d'une synchro dans l'espace,

j'avais tout un problème de synchronisation, parce qu'une fois qu'on a mis l'image comme il faut et que l'illusion se fait, par contre dès qu'un avion la traverse, si les deux images ne sont pas synchros, l'avion va disparaître d'un côté et réapparaître de l'autre. (Demo).

-Qu'on reste sur le projet E. ou qu'on remonte à des projets plus anciens, vous auriez d'autres problèmes concrets à évoquer, sur les données de capteurs à intégrer des IHM ?

-Ce qui me vient, c'est la vitesse de traitement aussi. Ce que je faisais, c'est des calculs de latence, de moyenne, savoir si typiquement… nous on fait de la remote tower de Muret, les gens disent oui ça semble permettre le contrôle à distance, mais il y a combien de décalage entre ce qu'on voit et la réalité? Typiquement, donc pour tout traitement, il faut avoir une idée, calculer des max et des min de latence. Il y a des normes dans l'ATC. Pour un radar il y a une certaine tolérance, pareil avec les caméras. Ce sont des calculs à la main, c'est des calculs qui ne sont pas possibles, tant qu'on n'a pas de t0. Pour Muret, on utilise le MTP, quelque part on a la même base de temps.

-Vous étiez amener à collaborer beaucoup avec ceux qui travaillaient au niveau de l'IHM? Des problèmes pour intégrer la vue panoramique ?

-Pas trop justement, c'était chacun sa partie, il n'y avait pas tant de collaboration que ça. La solution que j'avais choisie, c'était de faire des vidéos en fait. J'avais mon interface, ma 3D et une fois que c'est fait je génère une vidéo, donc une vidéo standard et n'importe quel lecteur peut la lire. Une widget dans Djnn (démo); ça c'est du Qt QML, là sur l'IHM t'as un un effet de flou et rectangle qui est positionnable avec le curseur, c'est du Qt pur là, c'est Muret en direct sur la vidéo, et l'idée, c'est de définir deux rectangles qui vont être utilisés pour faire un calcul de la luminosité moyenne. Pourquoi? Je pense que ça se voit, l'image de gauche est plus sombre que l'image de droite, pour la simple raison que le soleil est à gauche, ce qui fait que la caméra ferme un peu son oeil et donc l'image est moins lumineuse à gauche. Mais quand tu les regroupes pour en faire une seule, bah ça se voit. Pour harmoniser tout ça, je fais un calcul de luminosité moyenne des rectangles qui sont là et après je fais la différence des moyennes des deux rectangles, je divise par deux et j'ajoute/enlève la différence. Et je fais ça sur la totalité de l'image. Ce qui est marrant c'est que là c'est du QML. Du coup c'est deux widgets, widget droite et widget gauche que j'avais fait comme ça, et ça se fait en 120 lignes, et t'as un widget video qui est hyper simple. En gros, tu lui balances une video et t'as une widget video en QML qui fait déjà tout et t'as un widget pour le blur. Pourquoi je te parle de ça, c'est parce qu'en fait t'as des widgets qui sont vachement pratiques pour faire du vidéo, sauf qu'une fois que t'as fait ça, t'as toujours pas synchronisé. T'as pas accès au bas niveau, t'as aucun moyen de gérer tes timings. Parce que t'es haut niveau et que tu as perdu toute cette finesse de pouvoir lire la vidéo directement. Ce que j'avais fait à la main, je m'étais demandé si je pouvais le faire en QML, mais en fait y a pas moyen d'avoir accès. Plus un langage est haut niveau, plus on perd cette finesse du bas niveau, pour permettre la synchronisation des flux. Du coup je me suis pas embêté: je créé un fichier vidéo, je récupère le flux directement et le flux vidéo généré, c'est une entrée pour autre

chose. L'IHM vient après, je sépare les deux problèmes, je ne suis pas capable de gérer les synchros direct depuis l'IHM. Je les gère dans une vidéo en amont et l'IHM je la fais derrière.

-Et il y a d'autres langages qui auraient pu faciliter l'intégration de la vidéo ?

-Je pense que quel que soit le langage, c'aurait été un problème. J'ai déjà fait de l'incrustation vidéo dans des applis, mais par contre t'as jamais accès à ce qui est établissement de frame, ça tu ne l'as plus, et le t0 que tu peux avoir, tu l'as qu'en RTSP. Si t'es pas assez bas pour avoir cette info… L'abstraction fait que tu ne l'auras pas en haut. C'est un truc très particulier qui n'est jamais remonté dans l'abstraction. Et les frames, quand tu es assez bas niveau, t'as des timestamps, mais dans les lecteurs videos t'as juste "frame 1, 2, 3, 4" et tu perds toute la finesse de la précision de la frame. Un autre problème, c'est que les caméras ne sont pas vraiment en temps réel, le taux de frame il est variable.

-D'autres difficultés liées aux capteurs et au traitement des données?
-Après sur le projet E., on utilisait aussi l'ADS-B, donc en gros c'est l'avion qui envoie sa position par fréquence radio, donc en gros pareil pour un capteur ADS-B, là y a une base de temps, et là pour l'ADS-B c'est le temps GPS, c'est pas le même chose que le temps NTP. Et nous à l'époque sur le projet E., on avait calé le NTP de Muret sur le GPS et pas sur celui du labo et fallait que les données de l'ADS-B soient synchros avec les vidéos. Chaque monde est différent, et le LiDAR (autre projet) c'est encore autre chose, je me demande même si dans le LiDAR il n'y a pas de temps du tout et qu'on considère pas que le temps, c'est le temps auquel on reçoit les données, parce que c'est assez direct. Les frames forcément ça prend du temps, entre le moment où le capteur voit l'image et où le flux est généré, forcément il va se passer du temps. Y a forcément un temps non neutre de lecture. J'avais calculé une latence de 100 milli, ça ne peut pas paraître beaucoup mais en fait c'est énorme par rapport au déplacement d'un objet sur une piste, ça finit par faire 3m/s. Pour ces histoires de capteurs, pour finir, on utilisait des filtres de Kalman, en gros on agrège les data et on en sort des data lissées. Parce que l'ADS-B y a une tolérance à l'erreur, pareil pour la caméra, donc on regroupe les data pour réduire les erreurs, pour aussi avoir des trajectoires plus fines et plus lissées. Le but étant en sortie de capteurs quelque chose de clean. Mathieu pourrait t'en raconter plus. Y a la même chose pour les radars. Tout ce que je te dis là, c'est vraiment tourné vers l'événementiel. Par rapport au debug de tout ce qui est événementiel, c'est le gros défaut, c'est à la limite des outils de debugging classique. Y a plein de cas, où dès qu'on fait de l'événementiel, le simple fait de regarder, typiquement mettre un log, peut changer le comportement. Dans ce cas là, il faut avoir des logs, enfin genre typiquement PyTimechart, c'était de sondes sur le noyau, très light en fait, mais on n'est jamais à l'abris, quand on cherche pourquoi quelque chose ne se passe pas bien, au niveau de l'événement qu'est-ce qui se passe, ou dès qu'on rajoute des sondes ou logs, ça peut se remettre à marcher parce qu'on a changé les conditions.

-Quelles solutions vous imaginez dans ces cas-là ?

-Alors je sais que déjà le truc à ne pas faire et que je faisais, c'est de rajouter des timings à la main, des temps d'arrêt, comme ça je m'assure que le truc est lent et donc j'essaye de voir si une fois lent ça revient ou pas. Est-ce que c'est un problème qualitatif? Quantitatif? Est-ce que c'est un problème d'événement? Ou quelque chose de bas niveau? Souvent quand on ralentit tout le truc et que ça marche, c'est que c'est probablement un problème de timing. Après y a le problème de la reproduction. Ca me rappelle chez N., souvent on a des bugs, où y a pas de logs, pas d'erreurs claires et il faut un petit côté pif au mètre, pour savoir ce qui induit le problème. Et là on est dans le qualitatif. C'est un peu long mais si tu as 5 min, je peux te parler d'un bug qui nous était arrivé et qui était assez fin, pour t'expliquer ces histoires de reproduction et de qualitatif. Donc on faisait des tests à Toulouse et on faisait des tests aussi en Chine, à peu près les mêmes tests, sur le même téléphone. On faisait des tests toutes les nuits. Et les Chinois nous rapportaient que des téléphones étaient inaccessibles le matin. Donc en gros le téléphone, il est branché à l'ordi par USB, on utilise l'USB pour lancer directement les tests sur le téléphone. Le téléphone est vu comme un device USB sur lequel on se connecte. Nous, on n'avait jamais ce problème. C'était en phase de développement, donc des bugs y en avait partout, donc on n'était pas en train de regarder ces trucs-là. Puis les bugs partant, y avait toujours ça, toujours le même truc et nous (à Toulouse), on ne l'avait toujours pas. Alors on a pris le problème à l'envers: qu'est-ce qu'on fait qu'ils ne font pas là-bas? On regarde leur matos. On vérifie qu'on fait bien les mêmes tests, 3G, 4G. Nous on les fait en background, on ne se sert pas du téléphone pour passer un appel en 4G, on envoie le numéro par l'USB, en gros on va faire une commande. Les Chinois, ce qu'ils faisaient, c'est qu'ils allumaient l'écran (enfin c'est automatique, c'est les scripts qui font ça), un click est généré en bas pour faire apparaître ça (une saisie de numéro), et ensuite ils tapent le numéro en envoyant des événements de touche. Donc eux, ils tapaient le numéro comme si quelqu'un t'appelait. La seule différence avec nous, c'est que eux, ils allumaient l'écran. A partir de ce moment-là, je me suis dit "on va faire un essai" et j'ai pris un téléphone que je branche, je fais du trafic sur l'USB et en même temps je vais faire un script qui allume et éteint l'écran. A partir du moment où j'ai fait ça, en un quart d'heure on avait le bug: l'USB était connecté et on ne pouvait plus aller sur le téléphone. J'ai fait des analyses USB, pour voir si on perdait des trucs, mais au final on ne perdait rien, c'était vraiment le téléphone qui ne traitait pas un paquet, et à partir du moment où un paquet n'est pas traité, toute la connexion s'arrête. Et faut carrément enlever le cordon et le remettre, pour que ça revienne. Et après j'avais toujours ce truc avec l'écran. Pour faire court: au final, les Chinois trouvent le truc qui se passait; c'est qu'en fait, quand l'écran s'éteint, on va mettre la mémoire graphique à zéro, et ça en fait, c'est un transfert qui a saturé le bus entre le CPU et la carte graphique. Et quand le bus était saturé, et notamment le paquet USB qui passait, donc quand le bus était saturé, le paquet était perdu. Normalement, on est censé pouvoir lire quand quelque chose ne passe pas dans le bus, mais là y avait un bug dans le hardware donc le bus était buggé et ne disait pas les trucs. Donc tout ça pour dire, qu'il y a plein de cas dans l'événementiel où il faut faire des tests en burinant. Genre saturer le CPU ou la mémoire. Et voir est-ce que ça a l'air de faire quelque chose, changer les conditions pour le mettre dans des conditions moins favorables? Tant qu'on ne comprenait

pas déjà pour commencer qu'il y avait un lien avec l'extinction de l'écran…personne ne va trouver. Si ce qui est visible et la cause profonde sont aussi disjoints, et si on n'arrive pas à isoler l'espace où ça se passe, on n'a aucune chance de trouver.

I9

01/06/21

-Je n'ai bossé qu'en entreprise, et alors mon domaine d'application, avant c'était le médica, l'imagerie plus précisément, puis j'ai changé pendant le confinement et suis maintenant dans une boîte où ils font du cloud et de l'application desktop. La première boîte avait déjà cinq ans quand j'y suis arrivée et le projet est encore en cours. Les clients : quatre hôpitaux, un labo de cosmétique. Les utilisateurs dans la nouvelle boîte, c'est Facebook notamment, dizaine de milliers d'utilisateurs.

-Comment tu travailles, tu as une stratégie pour préparer ou organiser le code? Après si vous travaillez en méthode agile, j'imagine que tu as des tâches très précises assigner, mais ensuite comment tu procèdes ?

-Alors dans la boîte actuelle, les tâches sont définies avant de commencer. On a un *product owner*, la tâche est totalement définie. Il peut y avoir des problèmes d'interaction mais très fines qui vont être laissées au choix. Là récemment, j'ai eu une toute petite question d'UX, c'était du détail. Genre on a un champ de texte qui contient des valeurs et avec un maximum acceptable ; à quel moment on affiche cette information et comment : sous forme de conseil, d'une erreur? Pendant que l'utilisateur est en train de taper ou pas ? Le détail du comment, ça c'est à moi de le résoudre.

-Les tâches sont définies, et on vous donne des bouts de code, ou déjà une architecture, et vous travaillez avec des *mockups* ? Enfin à quel point tu peux tout de suite te jeter dans le code ?

-Alors oui, souvent des *mockups*. L'architecture du code lui-même, ça je dois le discuter avec mes collègues. Dans la boîte précédente par contre, il n'y avait pas d'organisation donc j'étais beaucoup plus responsable de ce que je faisais. Il y avait eu deux phases : au début j'étais complètement en autonomie et après ils ont recruté un mec qui venait d'un cabinet d'UX et qui a pris la main sur la partie design et fournissait des mockups.

-On peut maintenant si tu veux zoomer sur tes projets les plus récents. Peut-être tu te souviens de moments où quelque chose a été particulièrement difficile à exprimer dans le code, ou alors un comportement que tu obtenais sur ton interface et que tu n'as pas pu expliquer tout de suite ?

-Alors oui clairement, j'ai eu un cas comme ça de comportement hier sur mon interface. En gros, le produit qu'on fait, c'est un émulateur Android. Et la feature sur laquelle je taffais, c'était de pouvoir choisir dans les paramètres des devices avant de les lancer, la rotation qui sera appliquée au démarrage. Et en fait, il y a cette notion d'interaction entre deux logiciels, assez distincts d'ailleurs ; d'une part, il y a le *launchpad* qui sert à gérer la flotte de devices, et le *player*, et dans chaque player y a un device. Et donc on règle les paramètres du device dans le launchpad, et ensuite le *player* se lancer et c'est lui qui aura la rotation et caetera. Pour un device qui a déjà été créé depuis un moment, je n'avais pas de souci à avoir la rotation qui s'applique comme je la demandais. Par contre, pour un device à son tout premier lancement la rotation posait beaucoup plus problème ; soit elle ne s'appliquait pas, soit si elle s'appliquait, ça pouvait ne pas être la bonne; soit, à partir du moment où on commençait à interagir avec, ça se mettait un peu dans un mode d'erreur, donc visuellement le *player* lui-même restait à l'horizontal, mais le device à l'intérieur pensait qu'il était à 90 degrés. Donc j'avais des incohérences, et j'ai passé une semaine à ne pas trouver. Et du coup, au final ce qui a été choisi de faire : on lui force une valeur, à la fenêtre elle-même (parce que du coup il y a toute une communication, entre la fenêtre d'affichage, la fenêtre de rendu openGL et après la ROM android qui tourne en arrière). Et donc du coup ce qu'on a fait : la fenêtre d'affichage et la fenêtre de rendu, on lui met une valeur de rotation, même si ca ne correspond pas à ce qu'il y a dans la ROM à l'intérieur. Ensuite, tant que la ROM nous renvoie une valeur de rotation qui est différente de celle qu'on a demandée, on l'ignore. A partir du moment où on a eu le bon résultat, on le prend en compte. Et en gros, la solution qu'on a eue, c'est que "si jamais l'utilisateur fait une rotation de plus, alors qu'on n'a pas encore reçue la valeur qu'on attendait de la ROM, alors il faut que la valeur cible soit mise à jour en fonction de ce que l'utilisateur vient de demander". C'est assez technique, c'est assez *backend* comme truc. C'était un problème de synchro entre le player et la ROM.

-Et comment vous l'avez trouvé et résolu ?

-Bah clairement par tâtonnement. En l'occurence, c'est le développeur senior qui est dans la boîte qui était en train de relire mon code et qui a trouvé un endroit où fallait que je mette ce comportement à jour.

-D'autres souvenirs récents, où il aurait fallu se mettre à plusieurs ? Ou des bugs récurrents, que tu reconnais ?

-Alors y a un petit truc en vrai, et je pense que c'est lié au QML. En gros, souvent on a des problématiques de *sizing* des composants imbriqués. Soit on cherche à ce qu'ils prennent la place qui est disponible, soit on cherche à adapter par rapport à un composant qui existe déjà, comme une barre de défilement. Et souvent on obtient ni

l'un ni l'autre. Et du coup par rapport au QML, c'est intéressant en vrai: y a plein de paramètres différents qui gèrent la taille en QML. Comme c'est surtout basé sur des bindings de propriétés, c'est un peu dur de trouver la raison d'une taille, sachant qu'en plus, au-delà des bindings, y a souvent plusieurs systèmes qui s'imbriquent, avec les layouts et caetera. Moi j'avais l'habitude du QML dans ma boîte précédente, et dans la boîte actuelle, ils avaient abandonné le QML parce qu'il y avait des trucs qu'ils n'arrivaient pas à faire. Ils avaient opté pour des QWidgets mais rebasculent progressivement sur du QML. Et du coup, comme j'ai de l'expérience, clairement y a des problèmes que eux et moi n'attaquons pas de la même manière. Généralement je trouve la solution.

-Et comment tu la trouves ? Tu as une méthode un peu systématique pour trouver ? Ou tu unifies la manière de déclarer les tailles ?

-Et bien souvent, j'essaye d'utiliser une seule manière de gérer les tailles en gros. Donc eux ils vont souvent être passés par des bindings de propriétés diverses, et perso je préfère utiliser les layouts, c'est en mode "t'essaye de grandir autant que tu peux, avec ces valeurs préférentielles, min et max". Ouais donc en gros j'unifie tout. Sachant que en même temps, il y a beaucoup beaucoup de composants, c'est un code qui est gros et y en a beaucoup que je ne connais pas. Du coup, si jamais il y a des composants qui fonctionnent déjà et qui sont utilisés, alors j'essaye de ne pas y toucher. J'essaye juste de toucher à un niveau assez haut.

-Et tu utilises des outils de visu ou debug? Par exemple de la visualisation de layouts ?

-Je sais qu'en front comme en backend, ça pourrait être dans mes habitudes. Mais j'en n'utilise pas. Souvent ce que je fais, c'est que je vais passer par des astuces genre pour voir s'il y a des chevauchements de composants : j'englobe mon composant dans un rectangle très très visible dans une couleur qui pop. C'est un peu de la bidouille clairement.

-Et tu en as ressenti le besoin parfois ?

-A la fois oui, et en même temps ce genre d'outils, tant qu'on sait pas qu'ils existent, on en sent pas le besoin.

-Et tu aurais des exemples de moments où tu as reparcouru ton code, parce que tu cherchais la réponse à une question "pourquoi" ?

-Alors j'ai des trucs pile comme ça. Surtout dans mon ancien taff. Il y a eu une suite de problèmes, je peux essayer de le décrire, après c'est pas moi qui as trouvé la solution. En gros, la problématique à la base, c'était quand le praticien regardait une lésion sur un patient, il avait besoin de rentrer les informations de où cette lésion se trouvait sur le corps. Au tout début, j'avais mis un champ de texte libre, parce qu'on avait besoin d'avoir un truc qui fonctionnait. Et donc du coup c'était bien mais il y avait des doublons énormes, avec des informations différentes, donc quand on recoupait les données, on n'avait rien d'intéressant qui sortait. Du coup, j'ai essayé de faire avec une arborescence du corps, sauf que ça n'a aucun sens. Après on est passé sur un outil qui existait déjà : un dessin

du corps qui était découpé par zone, et ce qui était intéressant, c'est que ce dessin, c'était vraiment par zone de la surface du corps, et non pas seulement par membres. Et donc j'ai dû trouver une solution, pour quand on survole l'image, la zone survolée soit identifiée niveau code, et c'était des formes totalement irrégulières, genre une partie de l'épaule. Et en plus certaines parties étaient zoomées, avec plein de sous-zones pour la tête, par exemple. Là pareil, je peux te parler surtout de la solution : ce que j'ai dû faire, c'est créer une image où chaque zone avait une couleur différente, et des lignes de séparation entre zones, et ensuite au survol on allait regarder la couleur qui était sous le curseur et comme ça on avait l'information de la zone. Mais c'était pas suffisant. Fallait aussi pour notre interaction un shader qui au survol aille colorer toute la zone survolée par le curseur, et passe en transparent les autres zones. La difficulté, là-dedans, c'était une difficulté de code: le shader était écrit dans un langage de shader, le code de ces shaders fallait que je le mette dans une string dans du QML et en QML toute la partie impérative est en JavaScript, et en plus les données qui venaient à la racine de ça, c'était du C++. Donc passer par quatre langages différents pour une interaction, dont un juste pour une string. J'avais aucune auto-complétion, aucune coloration syntaxique. Et sur cette interaction là, on a aussi rajouté la possibilité de sélectionner un point précis, notamment pour les cas où le médecin avait besoin de distinguer deux lésions dans la même zone. La localisation devait donc être beaucoup plus précise qu'une zone. Mais ça moi j'ai peu bossé dessus; les deux problématiques qu'il y avait, c'était : comment on stocke c ette donnée du point, et donc en fait on a mis des coordonnées entre 0 et 1 sur toute la hauteur de l'image; et comme c'était entre 0 et 1, ça permettait d'avoir un scaling sans problème derrière. Et ensuite il fallait qu'on puisse sélectionner ce petit point qu'on avait placé et retrouver la bonne zone correspondante. Là en fait la carte était affichée juste avec des shaders et une superposition de deux images (le calque de ce qui était sélectionné et survolé, et le calque des séparations). Et les points, c'était des objets QML posés par dessus. Donc fallait que les systèmes de coordonnés correspondent bien entre ces trois trucs.

-Tu as d'autres exemples liés au fait de jongler avec plusieurs langages et les faire communiquer ?

-Je sais que j'ai eu plusieurs fois la problématique de devoir gérer les modèles qui sont côté C++ et que j'utilise côté QML, sachant que le QML fait des conversions de type automatiques, mais qui sont pas tout le temps là, c'est chelou. Il fallait gérer les types de la librairie standard C++ sur les gestions de listes par exemple, mais aussi les types utilisés par Qt sur les containers, sur les listes. Souvent du coup ça passe par les QVariant et sachant que les propriétés qu'on écrit en C++ pour le QML, ça doit être des dérivées de QObjects, mais que les listes ne sont pas des dérivées de QObjects, mais qu'elles peuvent quand même être des propriétés. Ensuite, ça c'est converti en JavaScript derrière. Et un autre truc relou que j'ai souvent, côté QML, c'est que les objets JavaScript ne respectent pas forcément le standard actuel de JavaScript. Donc du coup, je vais avoir certains objets qui ont des fonctions en moins, et d'autres qui ont des fonctions en plus. Parfois ça peut me poser problème, quand tu cherches à savoir comment faire telle chose. Et parfois je trouve une solution

qui marcherait avec du vrai JavaScript, mais pas avec le JavaScript de QML.

-Est-ce que depuis la fin du master tu as utilisé des FSM ?

-Alors j'en ai utilisé un peu, surtout pour moi, pour ma propre réflexion, pour comprendre certains comportements, pour comprendre ce que je cherchais à obtenir. Mais souvent les équipes de développement, soit ça ne leur parlait pas, soit ça aidait pas forcément. Je l'utilisais moi pour mettre au carré. Typiquement, si j'ai une problématique complexe dans la tête que je n'arrive pas à synthétiser moi-même, je vais chercher à poser la question, je vais réaliser que je ne sais pas poser la question à l'écrit. Donc je vais commencer à faire des graphs pour expliquer ça. Souvent en vrai, c'était pas des FSM carrées. J'utilisais une sous-partie du modélisme, ou je mélangeais avec des trucs qui viennent plus de l'UML. Je mets des trucs codes aussi dedans, je mélange. Et en général, ça me donnait la réponse.C'est arrivé qu'on utilise quand même une FSM que j'avais faite, je peux essayer de te le retrouver.

-Et donc ça t'aiderait de pouvoir avoir travailler dans ton éditeur avec une FSM graphique ?

-Je ne sais pas trop en vrai. En tout cas, pas un truc qui génèrerait du code à partir d'une FSM, parce que je fais pas confiance au code généré. Souvent parce que je ne vais pas le comprendre et le modifier ça va être compliqué. Sinon j'ai un autre truc à te montrer. La problématique, c'était: on a un process d'*upgrade* de device, et il y a plein d'étapes successives dans l'*upgrade*. Et il faut qu'on puisse, si une des étapes ne fonctionne pas, remettre à l'état avant toute modification. Et en gros, il s'est trouvé qu'il y avait un cas extrêmement particulier, où on pouvait avoir…attends d'abord faut que je t'explique ce qu'on faisait avant: on créait une copie du device, on mettait à jour toute cette copie, et à tout moment, si jamais pendant la mise à jour il y avait un problème, on faisait juste sauter la copie. Sauf que du coup, à la toute fin du process, l'idée c'est qu'on supprime l'original et on remplace par la copie en renommant l'original par le nom de la copie. Et si jamais il y avait un problème dans ce renommage particulier, ce qui allait s'appliquer, c'est qu'on avait déjà supprimé l'original, et s'il y avait un problème alors on supprimait la copie, et donc résultat il ne restait absolument rien. Il y avait quasiment aucune chance que ça se produise, parce que réussir à faire tout le process, réussir à avoir les accès en lecture et écriture, modification sans aucun souci, y compris au moment de supprimer le device original, mais que le renommage ne fonctionne pas…c'est très improbable que ça se produise. Et donc du coup ce que j'ai fait comme process: j'ai mon device original, je le clone dans un backup, j'upgrade, sachant que pour faire la version upgrade il fallait créer un nouveau device, et y mettre les propriétés de l'ancien device. Et donc du coup, je clone dans le backup, je supprime l'original, je créé mon nouveau device, je lui transfère cettepropriété, à tout moment, s'il y a un problème, je supprime ce nouveau device. Et surtout, j'essaye de remettre le backup à son nom d'origine. Mais si ça ne marche pas, c'est pas grave, parce que j'ai quand même un backup existant et au moins les données ne sont pas perdues. Là du coup, ce qui était compliqué, c'était de comprendre comment ça se faisait qu'on pouvait perdre des données. Et donc du

coup j'ai dû faire des schémas pour comprendre, je les ai montrés à mes collègues. Je crois que j'ai un dernier problème à "pourquoi" aussi. Ca va être plus facile. Alors c'était dans la problématique d'UX dont je te parlais au début, avec les gestions de couleurs et gestion d'erreur. Un autre problème, c'était que les champs de texte réagissaient super étrangement, ils ne réagissaient qu'après deux appuis, pour que le changement soit pris en compte à l'intérieur du champ de texte. Du coup pour ça j'ai fait un schéma, genre diagramme de séquence. Il y avait plusieurs bindings sur les valeurs qui étaient contenues dans ces champs de texte et du coup tout ça c'était connecté sur l'événement *text_edited*, et donc au fur et à mesure qu'on faisait nos modifications, ça les envoyait dans le backend et le backend les renvoyait dans le truc. Et donc le seul moyen pour qu'un changement soit pris en compte, c'est que pendant deux fois de suite la valeur ne change pas; c'est pour ça qu'il demandait souvent d'appuyer deux fois.

-Et ça, vous l'avez trouvé comment ?

-Alors le fait que le problème existe, ça a été découvert par une équipe QA qui passait tout le *soft* tout le temps. Et le pourquoi, ça c'est en examinant le code et en sachant, par mon expérience, qu'il y a ce genre de problème, qu'il faut se demander quel signal on utilise, quelle valeur on utilise dans le traitement du signal. En fait, souvent en QML, les signaux nous filent la valeur du signal qui vient d'être changée. Et là ce qui se passait, c'est qu'on se connectait à un signal, et on utilisait pas cette valeur-là, mais une valeur qui n'était pas synchronisée.

-Et t'avais un moyen de trouver ce problème sans expérience ?

-Ca l'aurait fait, si j'avais fait afficher, au moment où je reçois le signal, toutes les valeurs de ce composant-là.

-Dernière question large pour finir, en termes d'outils pour mieux comprendre ton programme, est-ce que tu as déjà pensé à des choses, ou ça pourrait des informations que tu aurais aimé avoir très accessibles pour résoudre certains problèmes?
-Je pense qu'en général, je n'utilise pas assez les outils de debug, parce qu'en général il faut, avant de lancer le debug, dire quelle information on cherche, et réussir à l'arrêter où il faut. Clairement la plupart du temps, je passe par des logs textuels, des valeurs qui m'intéressent. Mais je sais que l'équipe QA pour créer leurs tests automatisés, ils utilisent un outil, Squish, où ils lancent le logiciel, peuvent le mettre en pause et pointer avec la souris un composant, et Squish va être capable de leur donner les propriétés du composant. Et ça, c'est un truc que je trouverais intéressant d'avoir, pas juste pour les tests des interfaces graphiques, juste pour pouvoir inspecter les valeurs de tel et tel composant sur l'interface quand on code, au survol avec la souris.

I10

01/06/21

-Je suis principalement UX designer mais j'ai été amenée à plus coder sur un poste pendant le confinement, j'ai toujours été dans l'aéronautique ou bossé sur des drones, dans le privé, toujours chez l'entreprise T ou une start-up de T.

-Et comment vous travaillez, vous interagissez beaucoup entre designers et codeurs ?

-Alors je crois qu'au début, au *frontend*, ils leur passaient directement le code en tant que prototype, mais là on a changé depuis je suis arrivée, on est passé sur Figma et il y a beaucoup plus d'allers-retours entre nous, pour dire "ah bah du coup est-ce que ça tu peux changer, parce que ça va prendre beaucoup plus de temps", donc les estimations en termes de vélocité, elles changent un peu plus quoi.

-Il y a d'autres considérations que tu aurais sur les interactions entre designers et code ?

-C'est le challenge du métier en fait, on propose un design, puis un MVP (*minimum viable project*), on va essayer de faire le plus rapidement possible et être le plus efficace, donc si je propose un design mais que finalement il y a une librairie qui propose un autre design, je fais confiance aux librairies qu'il y a déjà, par exemple sur *React*, donc dans ce cas, je vais dire aux codeurs de faire exactement comme ils proposent. L'important c'est d'avoir l'*user experience* plutôt que l'UI parfaite.

-Comment ça se passe quand vous bossez ensemble? Tu dois leur fournir quoi exactement, des bouts de code ?

-Figma va leur fournir le CSS généré, et un peu d'HTML. Pour l'instant, on ne génère pas du code assez propre avec le logiciel pour que les développeurs le réutilisent derrière, mais tout ce que je produis, c'est un prototype sur Figma (c'est comme AdobeXD). Ce qu'ils vont voir c'est le flow de l'utilisateur et aussi l'interface vraiment comme elle est, haute fidélité quoi.

-T'as parfois à intervenir sur des problèmes de code ?

-Alors le choix que j'avais fait après le master, c'est vraiment partir que sur de l'UX design/ UI, et mettre le code de côté. Après dans mon entreprise, on m'a déjà demandé de repartir sur des tâches plus de code, et là j'étais toute seule et je n'avais pas une équipe de dév' et codais sur Unity.

-Et les fois où tu as eu ce type de tâches, qu'est-ce que tu devais coder exactement ?

-Là c'était un POC pour All-A-Lense, une réalité augmentée, et du coup sur Unity je devais développer les composants et les interactions entre composants. Donc quand on clique là-dessus qu'est-ce que ça fait, là on a besoin d'un menu pour l'utilisateur, c'était vraiment pour tester leur recherche derrière.

-Et dans cette expérience là, tu as des souvenirs de moments où tu obtenais des comportements inattendus? ou des bugs particuliers? Ou des problèmes simples mais récurrents que tu trouves presque typiques ?

-La plus importante galère en fait, qui est commune à mes deux expériences, parce que oui alors en fait avant T. et la start-up, j'avais fait un stage de master, c'était un peu similaire, c'était aussi de la réalité augmentée, je devais faire une carte en 3D et donc dans la start-up drone pareil quand je travaillais sur All-A-Lense, et les mêmes problèmes sont arrivés. C'est-à-dire qu'il y avait des problèmes avec Unity et les positions. Donc en fait t'as des positions en pixel sur Unity, la position en pixel parce que c'est de la réalité augmentée, et t'as une troisième position, c'est les données cartésiennes. Donc fallait tout le temps gérer les trois positions et pour vraiment bien comprendre tout ça, et vraiment faire un code où les objets se positionnent au bon endroit. Un problème, c'est que parfois sur l'interface il y avait écrit des positions, x, y et z, et il fallait remonter chercher l'information, pour savoir si c'était les pixels, si c'était la position globale ou locale (c'est comme ça qu'ils appellent ça) et parfois c'était pas très clair. Dans l'interface Unity en fait, tu peux manuellement écrire tes positions et ensuite dans ton code tu peux aussi changer les positions. Et en fait, les x et y de l'interface c'était pas évident si c'était global ou local. Donc je ne savais pas quel paramètre je faisais varier exactement.

-Et comment tu procédais pour réajuster ?

-Ah bah je prends un papier un stylo, ok, je fais le petit calcul, est-ce que c'est cette position, cette position et tout?

-Tu as d'autres exemples comme ça, où tu devais tâtonner et repasser par du papier stylo ?

-Bah après y a aussi les dessins avant de faire l'interface, en tant que designer, on fait souvent les protos papier, et c'est vrai que par contre avec le 3D y a pas beaucoup de logiciel pour faire des prototypes. Mais c'est pas en lien direct avec ta question. Attends je réfléchis. Après j'ai eu surtout eu des problèmes, mais c'était plus parce que j'étais à un poste pendant 6 mois pendant la crise covid où je n'avais pas les compétences nécessaires. Je ne comprenais pas comment faire le code, parce que le langage était nouveau pour moi.

-Et pendant ces 6 mois ou ton stage en réalité augmentée, est-ce que tu as utilisé des outils non classiques de debug, enfin autre chose que logs, prints ?

-Alors je faisais vraiment du debug encore moins que classique, des pauvres logs dans le code. Des moments je ne mettais même pas des points d'arrêt, mais juste des messages à afficher.

-S'il y a d'autres choses qui te reviennent sur cette période de six mois où tu as dû prendre en main le langage et le projet en cours dans la boîte, on peut aller creuser si tu en as envie, mais sinon on peut remonter à ton stage si tu préfères ?

-La plus récente, vu que c'est cette année, c'est pas mal. Donc je peux détailler là-dedans je pense. Je pense que y en avait beaucoup plus dans ma deuxième expérience plus comme designer, mais je peux trouver des exemples. Alors la complexité, c'est que j'avais un cercle plat, avec une certaine orientation, une certaine couleur, taille etc. Et tout ça c'était défini dans un fichier JSON à la demande du client, et donc dans ce fichier il y avait l'orientation. Et sur ce cercle-là, c'était un système de radar en fait: un cercle comme un camembert et dans ce cercle, il y avait des points à positionner dessus. Vu que tout est en 3D, ce plan-là, il avait une certaine forme et quand c'était plat, le point se positionnait avec la bonne taille et la bonne répartition dans les tailles, mais quand on changeait d'orientation, le point changeait aussi de forme. C'est comme une image quand tu la *resize*, elle peut être aplatie et déformée. Et là ça me faisait pareil quand je faisais l'orientation sur le cercle. Et là j'ai eu du mal à voir quel était le problème. Ma façon de trouver, c'est que j'ai tout essayé: changer les paramètres un par un. Et à la fin, j'ai dû transférer ça à quelqu'un d'autre et ce n'est pas moi qui ai résolu le bug.

-Tu as d'autres moments en tête où tu devais aller bidouiller comme ça, en faisant varier des paramètres ?

-Y avait aussi des moments où j'avais du mal avec des images que j'utilisais, pour les formes en 3D, j'utilisais des .PNG, pour que je code vite, donc j'utilisais des raccourcis: au lieu de créer une forme, genre une flèche, avec tous les traits, avec un super logiciel de design, je prenais direct une image. Et ça des fois ça me posait problème, au niveau de la taille, et j'allais manuellement changer sur l'interface, j'ai un peu triché en changeant la taille du contenant.

-Tu peux ouvrir du code pour l'avoir sous les yeux, ça pourrait te rappeler des choses ?

-Attends je peux réouvrir un logiciel de code là. Parce que c'est le genre de problème que j'oublie et je me souviens que c'était une galère, tiens par exemple l'interface de Unity. Des trucs contraignants pour moi, c'était déjà le setup de Unity, très long, plutôt le setup de Unity avec All-A-Lense, y a plein de procédures à faire, de reconnecter les deux, le PC et All-A-Lense, Unity et Visual studio, déjà ça c'est frustrant, parce qu'à chaque fois que je recommençais un projet, fallait refaire tout ça. Pareil pour le debugger, pour lancer l'application, t'es obligée de la lancer sur All-A-Lense, sinon t'as la petite visualisation en 3D sur Unity, sauf que ça ne fait pas les mêmes choses; un des gros problèmes, c'est qu'on avait les points d'intérêt sur la 3D (tiens je te partage

l'interface Unity), et par exemple on avait ce truc vert affiché quand je lance ici, mais quand je le lançais sur All-A-Lense, ça fonctionnait pas, on ne voyait pas les points d'intérêt. Ca m'a posé une colle, je me suis demandé comment ça se faisait qu'on n'avait pas ces données-là. Et en fait à la fin, il me semble que c'était un problème de lecture de JSON: je récupérais mes données dans un fichier JSON, le fichier JSON était sur l'ordinateur, mais quand il s'envoyait sur All-A-Lense, il ne passait pas; du coup c'était difficile de savoir quel document passait ou non et pourquoi. Et en fait il fallait utiliser une autre méthode pour lire le JSON. Ah attends, les *shaders* sur Unity! C'est une galère pas possible. C'est ce qui fait les textures, un petit code qui fait comment ton objet il apparaît. Et t'en as qui sont sympas que tu peux récupérer sur internet, mais c'est un code assez spécial. Donc quand t'as un problème et que tu veux modifier ton *shader*, bah tu peux pas. Donc soit t'en choisis un autre et tu adaptes ton design, soit t'appelles un expert de ce code-là. Par contre, c'est facile à insérer dans le code C++. Mais si jamais ça ne marche pas, tu ne sais pas pourquoi. Ca ressemble à du code CSS. Et c'est presque impossible de bidouiller. Tout ce que tu sais, c'est que quand ton *shader* marche pas il apparaît en rose sur son interface. Après c'est peut-être juste moi qui maîtrise pas assez.
Sinon, autre truc chiant (interface Unity, sélection de la vue de la caméra): ici, il n'y avait pas de UNDO. Tu mets le y, ça te mets la vue de top, tu cliques sur z ça te met sur le côté. Et en fait, tu ne pouvais pas revenir dans l'axe par défaut, qui est lui un peu en diagonale. Après en stage, c'était plus loin, mais j'avais eu des problèmes aussi, mais c'était Unity aussi. Le update s'update toutes les secondes, mais parfois tu te trompes et ça te fait des loops infinies. Ca, ça te met le code au démarrage, et ce bouton là, t'affiches toutes les frames, et ça dépend de ton setup, mais si tu te trompes là-dedans…tu peux te retrouver à faire du debug et ne pas comprendre pourquoi un milliard d'objets sont créés, alors que c'est le rafraîchissement de la fonction update… Et c'est assez commun de faire des erreurs comme ça. J'avais aussi des problèmes dans la structure de mon code. Donc ça c'est le script que tu as pour tous tes composants Unity; et du coup dans start, tu créées tes fonctions, variables, et il y a des choses que je créé aussi dans le start pour un composant et d'autres que je créé pour d'autres composants, et en fait j'avais des problèmes de concurrence, du coup il y en avait un qui voulait utiliser un objet, mais l'autre n'était pas encore créé. Les starts ne se mettaient pas en même temps, donc je devais faire un troisième truc, c'est dire "lui il est créé en premier, lui en deuxième" etc, comme ça celui-là peut faire appel à celui-là sans problème. C'est des erreurs communes, c'est peut-être moi qui est nulle après. J'ai eu un autre problème de *shader* aussi: je mettais mon PNG dans le material, et je voulais modifier la couleur de l'interface, sauf que mon PNG était rouge et si je modifiais du couleur via l'interface, tu mettais du vert et ça devenait noir, et je ne sais plus pourquoi. Mais du coup, j'avais dû créer des PNG verts et rouges pour ne pas avoir à modifier la couleur via le code. Encore une fois, j'ai dû trouver vite une solution facile, aller au plus rapide, sans forcément avoir le temps de comprendre. Donc dans le code, j'ai fait une bidouille genre, "prends ce matériel-là quand tu dois mettre vert, prends ce matériel-là quand tu dois mettre rouge", ça rajoutait des lignes de code, c'était vraiment pas du beau code, mais c'est pas grave. J'ai pas eu l'explication du pourquoi: est-ce que c'était un fonctionnement normal et juste moi qui ne savais pas l'utiliser ou

alors s'il y avait un problème avec mon PNG? Bon après j'ai des souvenirs toujours d'oubli de librairies, mais ça la console te le dit direct donc bon. Après ce qu'il me manquait aussi, ce sont des bonnes pratiques: créer son contrôleur, contrôler un truc pour les images; alors forcément tu fais juste des trucs à la vite fait; et ensuite t'as des problèmes de synchronisation, et tu réalises "oh mince c'est vrai que ce composant-là marche avec celui-là", "oh je n'aurais pas dû faire comme ça", tu te retrouves avec des liens assez compliqués avec les scripts.

-Et justement comment tu retraces ces liens ?

-Alors sur Unity y avait un truc pas mal. Par exemple, tu passes ta souris sur main_player et tu fais control, click et ça te renvoie sur la fenêtre où le composant est créé, puis ensuite tu peux récliquer et ça te renvoie sur l'autre. Et au final ça te permet de mieux comprendre ce qui s'est passé, de faire les liens. Ca te fait une sorte de chaîne.

-Est-ce que sinon il t'est déjà venu l'idée d'un assistant qui serait cool ?

-Moi je rêve de faire ça avec mon proto, avoir plein de wire frames, quand je clique dessus tout ça ça marche, je peux faire mes tests utilisateurs, et appuyer sur hop "développer" et ça fait du code, plus de code généré et qui soit vraiment bien quoi. Quitte à faire une petite vérification après, parce que ça prend tellement de temps aux dév' le front-end et ils aiment pas ça; ce mois-ci ils sont full-stack pas que front-end, avant ils étaient back-end. Limite ça pouvait être généré juste le front end, les boutons, les composants, les choses qui deviennent bateau et qu'on réutilise. Après t'as des outils qui t'aident à faire cette transition; par exemple, *Storybook*. T'as tous tes composants UI, et t'as le code et le CSS, comme ça t'as juste à copier coller. Genre tu veux un bouton et un drop down, et tu récupères le code de chaque et puis voilà, tu les mets à côté. Après, ce que j'aime bien, c'est quand t'appelles tes fonctions, que tu n'aies presque rien à écrire, un max d'auto-complétion avec du "tab tab tab" et ça tombe tout tout seul, t'as juste à modifier ce qu'il faut, de pouvoir aussi faire un max de copier-coller. Pour moi, le métier de dév' c'est ça au final: tu fais confiance à ton outil qui va te mettre les bonnes choses au bon moment, et aussi parce que si tu utilises les bonnes pratiques normalement, c'est comme ça que c'est fait. Tu mets un get, il te propose un set. Après, parce que c'est la bonne pratique de le faire. Limite, t'as pas besoin d'écrire. Pour moi, un mauvais outil, c'est celui qui ne voit même pas que tu viens d'écrire une faute, vraiment pas d'aide à la reconnaissance des mots.

I11

01/06/21

-J'ai eu plusieurs domaines d'application : automobile, digital health, technologie haptique, reconnaissance de gestes. Et toujours dans l'académique.

-Tu te lances direct dans le code, comment tu travailles, mockup ?

-Ca va dépendre beaucoup du projet, j'ai beaucoup codé pour des expés dans mon parcours, donc je développais surtout la technologie qui permet de connecter les données, donc c'était en mode projet avec un vrai produit. Et je sais comment on fait, j'ai l'habitude de ces expés académiques, donc je gribouille juste un peu, je réfléchis à quelles sont les données qu'on doit récupérer, à quel moment on doit les récupérer. En fait, je suis souvent une approche modulaire, je conçois différents modules et cherche à les faire communiquer entre eux.

-Tu fonctionnes qu'avec du code textuel ou tu fais parfois des allers-retours à des représentations graphiques du code, dispo dans des outils de debug par exemple ?

-Attends laisse moi réfléchir. Quand je regarde le code des autres, j'utilise de diffs pour des retours avec des couleurs. Par contre, sur un des projets en licence, on utilisait des graphes, ils avaient créé, là où j'étais à l'université de Toulouse, une approche graphique qui représentait des composants pour représenter le data flow et les événements. Mais c'est la seule expérience que j'ai de représentation graphique de code.

-Et est-ce qu'on pourrait aller zoomer sur des problèmes précis dans des projets…question large pour commencer : est-ce que tu as des exemples de moments où tu as eu du mal avec la représentation textuelle du code et a dû passé par un gribouillage ou quelque chose pour te représenter les choses ?

-Alors ouais, alors par exemple, dans un de mes projets automobiles, on avait une expérience. On a construit un simulateur et les données qu'on voulait récupérer, c'était du eye-tracking, l'utilisateur devait manipuler une liste pendant qu'il conduisait, il y avait des cibles sur la liste, choisir le bon élément etc. Donc fallait récupérer les performances sur la liste, les performances de conduite et où il regardait. Donc c'était un système assez complexe, il y avait énormément d'inputs et d'outputs et des modules complètement différents qui n'étaient pas intégrés, chacun indépendant. Donc pour moi ce qui était difficile, c'était de dire qui fait quoi et où les données vont et comment on récolte tout, pour en fait les récolter au même endroit, parce qu'il te faut synchroniser toutes tes données, pour pouvoir avoir des mesures qui sont cohérentes. En fait, j'ai eu le même problème sur SAM, le truc des poupées, parce que c'est pareil. On enregistrait le visage des enfants, la façon dont il jouait, il y avait aussi les poupées qui étaient informatisées et indépendantes aussi donc il fallait récupérer ce flot de données là. Donc en fait

de nouveau le challenge c'était, comment tu synchronises tout, pour avoir des mesures qui correspondent au comportement de l'utilisateur.

-Et là comment tu vérifiais que ça se synchronisait ? et est-ce que trouvais ça satisfaisant ou tu aurais eu besoin de quelque chose pour rendre ça plus efficace ?

-J'essayer de synchroniser les horloges de chaque système et ensuite je vérifiais les timestamps. Avoir plus facilement accès aux timestamps peut-être. Dans tous les cas, les timestamps me permettent une fois que je les ai, c'est un moyen de post-analyse de rectifier, recalculer, re-synchroniser a posteriori. De savoir si c'est synchronisé ou pas. Il y a une grosse phase de tests avant de lancer l'expé en tant que telle, surtout que nous on travaillait avec des enfants, donc en fait ton truc doit être béton avant que tu arrives à l'école, en plus c'est imprévisible les enfants, comment ils vont jouer avec ton truc. Par contre, on avait pas trop de moyens de vérifier à la volée ; le truc c'est qu'on avait un très gros flux, parce qu'on enregistrait de la video HD, on avait des ralentissements, du gros travail pour l'écriture sur les disques; donc on pouvait pas regarder en temps réel. Fallait toujours faire une petite session et vérifier a posteriori ; et c'était un peu relou.

-D'autres problèmes que ceux de synchronisation pour faire communiquer tous ces éléments ?

-Il y avait aussi des problèmes de batteries, s'assurer que tous les capteurs marchent tout le temps, on utilisait en plus des technologies innovantes, un peu en bêta quoi. Du *real sense* avant que ce soit distribué, du coup parfois les pilotes étaient pas super oufs et ça buggait. Quand on oubliait de recharger les batteries, plus de synchronisation et plus de données. Donc on avait aussi ce problème d'énergie et de reliability du hardware.

-Et au niveau du code, ça posait des diffractés spécifiques, des choses difficiles à exprimer dans le code, ou tous les problèmes étaient autour du code ?

-Je crois que l'optimisation de l'écriture sur le disque, ça m'a posé problème, mais c'était pas du code, c'était plus moi et l'approche du problème. Alors sinon pour l'expressivité du code, je réfléchis… Là où on va être limité nous, c'est dès qu'on veut créer de nouvelles interactions, là tu dois souvent faire des passes-passes quoi. J'essaye de te trouver un cas précis. J'ai pas eu ce genre de problème sur le dernier projet, vu que la techno existait déjà, c'était juste pouvoir mettre des briques en place. Après pour moi la difficulté, c'est quand fallait faire des trucs que je connaissais pas, plus quand je sors de l'habitude de faire des expés. Parce que d'habitude, je n'ai par exemple jamais à me soucier de l'optimisation du code, faut juste que ça marche, que j'aie les données. Mais quand il faut créer des nouvelles interactions…attends je réfléchis sur les autres projets quand j'ai dû créer des nouvelles interactions.

-Par exemple des cas où tu te serais sentie limité pour exprimer ce que tu voulais ?

-Alors ouais, quand on travaillait sur le premier prototype de ultra-haptics, mais après c'est pas que un problème lié au code lui-même. Donc nous on avait le tout premier prototype et il ne marchait pas bien, il fallait toujours trouver des passes-passes. Par exemple, la fréquence à laquelle tu sens le poing et le sens bouger, bah c'était beaucoup plus lent sur les premiers prototypes. Et la fréquence de réaffichage du tableau elle n'était pas aussi performante qu'auourd'hui ; il y avait un goulot d'étranglement et ce qui se passait, c'est que quand t'allais trop vite, le boitier chauffait trop et s'arrêtait. Donc il a fallu qu'on trouve des moyens d'optimiser la fréquence de réaffichage du tableau. Alors après est-ce que ça m'a posé un problème dans le code ? Alors ouais. Parce qu'en fait, y avait pas de docs, le code n'était pas clair, on avait quelques fonctions et caetera.

-Donc là c'était du réajustement par essai erreur ?

-Ouais, on tentait des trucs, ça marchait ça marchait pas.

-Et tu aurais aimé avoir un outil pour t'appuyer dans ce cas là pour simplifier les ajustements ?

-J'aurais bien aimé pouvoir connaître "étant donné la techno que j'aie, quelle est la limite à laquelle je peux aller". Tu pourrais avoir un curseur qui te change la fréquence à laquelle ça affiche le truc sur ton tableau, et dire voilà à partir d'un certain seuil ça ne marche plus. Et idéalement avoir un lien entre cette jauge et le code.

-Et tu as d'autres exemples ou tu as dû utiliser d'autres choses que des méthodes de debug classique, procéder autrement, passer par du essai erreur ? Avec des problèmes de devices physiques ?

-Bah j'ai envie de te dire que c'est toute ma carrière. Dès que tu fais une nouvelle technique d'interaction avec des capteurs, tu sais jamais, tu dois tester, ça marche, ça marche pas, tu reviens dans ton code.

-Et quelles sont les méthodes que tu vas utiliser et combiner (voire aussi du debug classique) pour y arriver ?

-Alors moi avec le temps, j'utilise de moins en moins le debug. J'ai pas besoin de produire du code propre, personne va le relire. Quand tu fais de la technique d'interaction, en fait tu veux que ça marche et t'as pas de compte à rendre. Du coup, les outils de debug tu les utilises quand ça compile pas, mais si la machine fournit la performance dont j'ai besoin, bah go. Par contre, tu veux, c'est vrai, que tes utilisateurs soient dans une situation contrôlée et que ça corresponde au mieux au comportement qu'on prévoyait nous. Donc le match attendu, il n'est pas tant au niveau de la performance, que dans ce match entre comportement obtenu et attendu. C'est moins le code que la façon dont les gens vont interagir avec ton système qui importe. Et quand j'ai un problème de capteur, là ma stratégie c'est pas du debug classique, c'est de me fixer des exemples, qui vont peut-être correspondre à une version simplifiée de la situation finale dans laquelle tu vas arriver. Si on prend le papier sur les gestes sur le ventre, tu regardes quels gestes tu peux faire, tu vas commencer par des gestes simples. Tu vas essayer de voir si tu peux reconnaître un

tracé vers le haut, il faut te mettre alors dans la peau du capteur, tu dois voir le monde comme dans le référentiel du capteur, comment il voit le monde ; donc si je fais un geste vers le haut, ça correspond à quoi pour le capteur; tu fais ensuite un match et tu vois si ça correspond. C'est du debug mais du côté humain quoi. Après un détail important sur ce que j'ai fait : la plupart des projets qu'on fait, c'est des projets jetables. On va coder très vite, la qualité de code c'est toujours un trade off avec la vitesse/ niveau de qualité et y a un curseur. Mais quand t'as 6 mois pour soumettre à UIST, tu te soucies pas de si ton code est optimisé, parce que tu vas le jeter. Ton code il montre juste une technique d'interaction, il risque de ne pas être réinjecté dans du software. Et puis y a un problème politique : tout ce que tu fais pour l'université ne t'appartient pas. Donc même si tu trouves la technique qui va changer le monde, tu pourras rien en faire parce qu'elle appartient à l'université.

-Tu aurais des exemples où tu n'obtenais pas le comportement que tu attendais ?

-Alors là récemment, j'ai des exemples en Smala parce que j'ai commencé la semaine dernière avec le langage. En ce moment, j'essaye de coder un *pie men*u. Et V. m'a aidé à debugger des trucs, il avait d'ailleurs eu les mêmes problèmes. Attends faut que je me souvienne. J'ai galéré sur des trucs pendant une journée, et V. pouvait régler ça en une heure. Alors plusieurs trucs : y avait la machine à états et la façon dont les calques sur mon SVG étaient fichus, parce que du coup je n'arrivais pas à voir les deux modèles et lier les deux modèles (le SVG et la machine à état): comment je dois façonner mes calques pour que ça corresponde à un état de la machine à état et que, quand il y a une transition, que ça m'emmène dans le calque que je veux.

-Et là le problème, c'était la structure du SVG, problème d'identifiant ou la FSM ?

-Donc oui la façon d'identifier les calques, et V. m'avait dit que tous les IDs doivent être différents, mais c'est pas tout à fait le cas, parce que ça dépend où ton calque se trouvent dans l'arborescence, ça dépend des parents. Donc ça doit être différent quand ils sont au même niveau. Après il y a la façon dont on écrit la machine à états. Avec une représentation graphique, il n'y a pas d'ambiguïté ; ou plutôt je veux dire: c'est lié au fait que c'est pour moi un nouveau langage. J'ai l'habitude des slots et Qt, et c'est vrai que c'est plus ou moins le même concept mais c'est des manières différentes d'exprimer la même chose. Plusieurs niveaux dans lesquels je ne suis pas encore expert pour être productif. Des FSMs j'en avais juste fait au master IHM, mais ça reste pour moi la manière la plus fiable d'exprimer un système interactif. J'avais utilisé SwingStates il y a très longtemps.

-Il y aurait des outils graphiques ou textuels qui t'aideraient, de la coloration ou de contexte dans le code à la représentation graphique ?

-Moi je rêve d'un outil où tu fais ta machine à états, tu la sketches. Enfin moi la chose que je fais toujours, c'est que le prends un crayon et un bout de papier, ou Omnigraffle, et je sketche mes états de mon système. Et j'aurais envie d'avoir un logiciel qui à partir de ça

“schtounk” te produit le code textuel qu’il faut. Parce que le truc des FSM, c’est que c’est pas un code qui va changer beaucoup. D’un langage à un autre, elles sont assez similaires, c’est juste le langage qui change, la logique reste la même. L’expressivité reste la même, même si la FSM gagne en complexité avec le nombre d ‘états et transitions.

-T’as ton programme et tu veux savoir qu’est-ce qui cause quoi dans ton programme, t’obligeant à reparcourir le programme, est-ce que tu auras des problèmes de ce type, où tu t’es posé une question similaire ?

-Alors j’ai beaucoup bossé sur tout ce qui est inférence causale. J’ai un petit exemple en ce moment. Donc je bosse en ce moment avec J. et H., et je devais essayer de faire marcher un bracelet avec des trucs haptiques. J’ai eu du code d’une stagiaire, du code H. et J. Et ça ne marchait pas, et j’ai dû essayer de comprendre pourquoi ça ne marchait pas. Et c’est là où ça devient intéressant : si t’as trois codes qui doivent communiquer ensemble et que tu ne sais pas d’où tu peux venir l’erreur…. La stagiaire faisait de l’Arduino, H. et J. faisaient du Python pour faire voler le drône et envoyer les données. Y avait deux choses qui marchaient pas: les messages qu’attendaient la stagiaire n’étaient pas les bons (on utilisait le bus Ivy); donc quand les messages arrivaient, H. n’envoyaient pas les bons messages; et ensuite il y avait une autre erreur, et ça je ne sais pas comment j’ai fait pour la détecter: les messages n’étaient pas cadencés à la même vitesse, et c’est l’expérience qui m’a fait trouver. La première chose que tu fais dans les Arduino, surtout quand tu communiques en serial, c’est que tu vas configurer la vitesse de communication. Et en regardant le code de la stagiaire, j’ai vu que la vitesse n’était pas la même que dans le code de J. Donc forcément c’était pas synchronisé et les messages se perdaient. Donc là quand t’as des codes hétérogènes, t’as besoin d’explorer la causalité. Parce qu’en fait il faut être assez expert pour trouver ce genre de problème, pour avoir une idée de ce qui peut être la cause. C’est comme quand tu vas chez le garagiste et que tout de suite il trouve, il te dit direct “ah oui mais c’est ça”. Donc un truc qui permet d’explorer la causalité, c’est hyper utile pour un débutant mais aussi pour quelqu’un qui doit reprendre en main le code de quelqu’un d’autre, ou doit prendre un main un nouveau langage. En fonction, du type de langage, c’est pas la même façon de penser. Tu vas avoir des causes différentes : par exemple, tu prends la programmation fonctionnelle, tu vas avoir des boucles, de la récurrence et là t’as un mauvais cas d’arrêt ; et dans la programmation impérative, le problème ça va être ta boucle. C’est des cas d’arrêt, c’est la même chose conceptuellement, mais l’expressivité n’est pas pareille, donc la cause dans le code n’est pas au même endroit.

-Est-ce que justement passer des langages plus impératifs que tu utilisais au début à des langages “déclaratifs” t’a posé des problèmes de compréhension de code ?

-Ah oui clairement, et encore maintenant. Dès fois je me casse la tête. Là par exemple après une semaine de Smala, je me suis que la meilleure façon de faire, c’est d’aller voir tous les *snippets* dans les *cookbooks*, sinon je ne vais jamais y arriver. Donc là je vais déconstruire mon pie menu et je vais me dire de “quelle brique”’ ou

quelle sorte de fonction ou mécanisme je vais avoir besoin, et ensuite je vais chercher ces mécanismes dans les cookbooks. Je pense que c'est l'approche que je vais suivre, sinon je vais y passer ma vie. Le *quick and dirty*, c'est la meilleure approche au début. Après je connais bien les concepts, similaires à ceux de slots dans Qt. Mais le problème pour moi, c'est que ce soit un nouveau langage. Les déclarations sont à l'envers, les *assignments*, les différents types de connexions, et dès fois c'est pas clair pour moi ; mais c'est au niveau de la syntaxe, qu'il y a des trucs pas usuels pour moi. Après rien à voir, mais j'ai l'impression que ce que vous faites, c'est du end-user programming, et je trouve que le danger des outils, c'est d'abrutir les programmeurs, tu vois ces outils sont bien quand tu as compris le concept et tu sais ce que tu fais. Aujourd'hui on apprend aux étudiants à coder en Python, alors que nous à l'époque on apprenait à coder en C avec des pointeurs. Aujourd'hui tous ces codes interprétés…c'est à trop haut niveau. Alors t'en as pas besoin pour vivre, tu vas en entreprise t'as pas besoin de savoir ce qui se passe en mémoire, mais c'est quand même vachement important de le comprendre et tu perds ça, que ce soit avec le ramasse-miettes ou toutes les stratégies qu'on a mises en place pour faciliter la programmation. Et en interaction, on travaille souvent avec des capteurs, donc c'est important de comprendre le bas niveau.

-A part les outils de debug classiques, les tâtonnements par essai-erreur ou juste l'expérience que tu évoquais pour trouver les erreurs dans ton code, tu aurais utilisé des outils moins familiers, des visu/représentations ?

-Ma qualité de code et ma connaissance des outils pour contrôler mon code, ça s'est arrêté à peu près à la Fac, il y a 12-13 ans. Après je n'en ai plus eu besoin. A l'époque, la mode c'était les tests unitaires. J'ai toujours travaillé plus par tâtonnement qu'avec des outils formels. Après je rêverais d'un truc de preuve pour les systèmes interactifs, pas du côté code, mais du côté humain. Est-ce qu'au lieu d'avoir un truc qui te vérifie ton code, on aimerait avoir plutôt un truc qui vérifie le résultat d'un code, d'une manière plus haut niveau. Par exemple si on reprend l'exemple du geste, savoir si c'est un geste vers le haut ou pas que j'ai codé. Avoir un programme qui vérifie pas : "ça, c'est ce geste". Vérifier si le comportement interactif est correct.

I12

02/06/21

-Comment ça fonctionnait dans ta boîte, entre codeurs et designers ?

-On faisait tout de A à Z parce que la structure était petite. Après on a eu besoin de designers, nous on s'occupait du design graphique des interfaces, conception du système interactif et gestion de projet. Si le projet du post doc compile (codé en PyQt), je pourrais te montrer un ensemble de problèmes récents. Mais faut que je le fasse marcher pour bien t'expliquer ce qui bloquait. Alors le contexte: c'est un simulateur de trajectoire dans un microscope électronique, et donc t'as l'affichage de ce qui se passe à l'écran et tu peux créer des coupes dans ta colonne. Ca veut dire en termes d'interaction tu vas avoir un découpage par partie, va falloir faire énormément communiquer les informations entre tout ça. Va falloir que là je passe les variables et que là je sois informé de ce qui se passe ici. C'est-à-dire que là si je modifie une valeur dans ma lentille C1, ça va mettre à jour tout, et aussi ici dans la simulation qui donne une vue plus détaillée, plus réaliste de ce qui se passe dans le microscope. Dans les autres vues on simplifie, car en réalité il y a d'autres électrons qui se baladent à l'intérieur, et on enlève les positions extrêmes des électrons (pour avoir un faisceau propre).

-Et comment tu gères la communication de toutes ces informations ?

-Donc là, c'est des vues, chaque vue est associée à un modèle, et tu vas avoir un méga modèle qui va permettre de faire communiquer. Et ça, c'est un code que j'ai repris, c'était codé en chef d'œuvre (projet de semestre) par des élèves du master IHM, et moi je rentre là-dedans et j'ai dû partir de ça. Et t'as un *main* qui va instancier tous les *plugins* dont t'as besoin, et qui va charger le modèle, et les différentes vues (les vues des coupes, la vue globale). Il faut se dire au début là "mais il charge ça, donc faut tracer, ok là c'est le modèle principale et après faut que j'aille là". Ils m'ont livré un pseudo-découpage de code, donc c'est quand même un peu structuré. Dans tes *components*, t'as les différents éléments qui correspondent à des positions dans ton écran. Et tu te dis, depuis la vue globale, qui va charger les symboles de la vue par exemples. C'est cette instantiation de hiérarchie de fichiers…franchement tu pleures. Tu traces, tu fais des prints pour voir "ah là c'est ici que c'est chargé", ou tu fais des erreurs dans ton code, en créant des bugs, pour que la console te disent dans quel fichier ça va pas.

-Et là dans l'éditeur, tu as pourtant pour les fonctions accès aux fichiers où elles sont appelées ou créées, non ?

-Tu dois pouvoir juste voir où une fonction est définie. Mais t'es quand même très peu aidé. Ca marche pour les fonctions, tu peux faire un "go to définition", tu peux aller au fichier où est déclarée la fonction que t'importe, mais t'es passé dans un autre fichier, donc c'est pas évident.

-Et qu'est-ce que tu voudrais à la place ?

-Ne pas avoir à quitter le fichier, qui permette de tracer, de me donner les liens de dépendance entre les fichiers. A la limite, j'ai une autre fenêtre ou une autre colonne où je pourrais me balader dans ces liens de dépendance. Parce qu'en fait là j'ai les fichiers qui sont stockés sur le disque dur, et c'est en fait pas super intéressant, c'est bien si c'est cohérent, mais si le mec a mal rangé ses fichiers, ça te dit pas grand chose. Par contre, si t'as une colonne avec tous les liens de dépendance, et que tu passes de l'un à l'autre: ça, c'est intéressant. Parce qu'en fait je suis un développeur avec une capacité d'abstraction très faible, j'ai appris le développement par le développement séquentiel, donc moi j'aime bien quand ça se passe dans l'ordre. PHP c'était ça.

-Donc tu n'aimes pas quand un comportement est éclaté sur plusieurs fichiers ?

-Ah oui, j'ai du mal. Autant je pense qu'un codeur qui fait de l'objet, de la programmation MVC depuis très longtemps, il s'y retrouve. Mais moi j'ai vachement de mal et c'est vraiment par habitude, en rentrant dans un projet que tu vas acquérir une connaissance de ce projet-là mais qui est éphémèr, lié au fait que tu travailles dedans. Le risque, c'est que si je me remets dans ce projet, je ne vais plus savoir, je vais devoir réacquérir cette connaissance. Il n'y a pas de trace.

-Et tu n'aimes pas qu'on te renvoie dans un autre fichier ?

-Non, parce que moi je vais travailler sur un fichier en particulier, et j'ai pas cette capacité à m'extraire de l'existence de la donnée sur le disque.

-L'info que tu voudrais afficher dans le fichier que t'examines, c'est donc savoir où la fonction a été définie, mais il y a d'autres choses pour lesquelles tu voudrais un accès direct ,sans avoir à sauter dans un autre fichier ?

-Où elle a été définie, et cette classe elle appartient peut-être à un modèle et j'aimerais bien dans le modèle savoir à quel niveau du modèle elle est. Il y a le risque de charger plusieurs fois un modèle. Parce que tu te dis "tiens, ce modèle il a la donnée"; mais par contre, au niveau dans lequel je suis, je ne peux pas l'appeler directement ce modèle parce qu'il n'est pas instancié, il a été instancié bien plus tôt. Donc comment je vais le chercher ce modèle qui a été instancié plus tôt? Faut que j'aille le rappeler avec le bon nom de la classe. Par exemple ici, on a le *trajectory_model*, celui-là il est instancié dès le départ, c'est lui qui permet de calculer les trajectoires. Et il est instancié vachement plus tôt et je ne sais pas comment aller le chercher. Et ça c'est galère, avec ce danger de réinstancier un modèle. Ca va marcher, mais c'est le truc à pas faire. Et tu vois dans l'éditeur, c'est difficile d'aller le chercher. Le *trajectory_model* va servir à générérer des signaux (c'est le côté interactif si tu veux), il écoute ce qui se passe un peu partout et il répercute, tous ceux qui sont abonnés. Et là c'est pareil: trouver ces liens… Comment je vais faire pour informer les autres vues ? Est-ce que je vais rappeler *trajectory_model* ici ? Ou alors je mets à jour mes vues sans passer par un modèle central. Après j'ai fait le choix

de continuer ce que les étudiants avaient fait, en proposant un modèle principal; mais là il y a des vues dans des vues qui appellent des modèles, et encore se pose la question de “dans quel niveau je suis? Et est-ce que je peux appeler ici mon signal dans ce niveau-là? Ou je dois rappeler trajectory_model à un niveau inférieur et passer par des couches intermédiaires? Le *trajectory_model* est là et donc lui ici global_trajectory, quand la vue paraxiale change il informe le *trajectory_mode*l. Bon et là je peux faire des “go to reference” mais c’est foireux et je finis par le faire à la mano, et je vais moi chercher *paraxial_view* dans *trajectory_mode*l.

-Et tu repères vite qu’une erreur est due au fait que tu t’es pas abonné à un signal au bon niveau ?

-Non parfois ça peut être dur à trouver. Soit tu n’appelles pas le bon modèle, soit tu ne l’appelles pas au bon niveau, et donc tu peux ou avoir une erreur ou alors il n’y a rien qui arrive, mais alors tu ne sais pas à quel niveau ta donnée se perd, parce que tu sais pas par où passe ta variable. Donc tu dois là aller faire du débuggage pour voir “ah là elle existe encore, là elle se perd, pourquoi elle se perd”. Bon au début je ne maîtrisais pas complètement les signaux sous Python, mais même quand tu maîtrises, quand t’es sur un gros fichiers, avec plein de fichiers, plein de vues, plein de modèles (bon après faudrait peut-être le repenser, en termes de design de code), c’est le bordel et t’es perdu. Quelqu’un qui doit rentrer dans le projet, il lui faut un mois pour comprendre, les interdépendances, où tu appelles cette classe, “qu’est-ce que dit ce modèle?”. Moi quand je suis rentrée dans le projet, je me suis dit l’approche MVC, elle te fait une grosse bouse quoi. Parce que tout est splité, tu n’as plus du tout de vision de lien, de dépendance, de comment les choses vont communiquer entre elles, quel type d’informations elles peuvent se passer.

-Tu as une stratégie pour reconstituer ça ?

-Bah c’est dans ma tête, en travaillant sur le code. Tu remontes le fil un peu du problème.

-Tu utilises des techniques de debug ?

-Les logs ou de l’écriture de données de debug. J’ai essayé au début de faire un truc un peu systématique, avec un diagramme de classes. Si ton code marche mais que par contre tu veux savoir ce qu’il en est de telle variable à tel endroit, là je mets un point d’arrêt. Mais c’est tout. Pas de graphe rien. Parce que c’est vrai qu’il me manque une capacité d’abstraction au développement.

-Et en plus de la difficulté à reconstituer le fil, il y a d’autres choses ?

-La partie 3D, mais c’est plus parce que tu t’arraches les cheveux quand il faut positionner des éléments graphiques 3D en Python, les axes de rotation, les projections, mais c’est propre à la 3D, et c’est pas la faute de la structure du code ni de l’IDE. Bon sinon quand j’ai lancé des threads supplémentaires. Parce qu’en fait cette application elle est connectée à un microscope, elle écoute en permanence; il y aussi un device de contrôle avec un bouton rotatif et un écran tactile,

qui permet à l'utilisateur de faire des manips sur l'application et le microscope. Et ce device coûte. Là c'est assez cloisonné ce lancement de thread en parallèle, pour des raisons de robustesse, donc t'as pas accès à tous les modèles. Tu peux pas aller chercher un modèle que t'as pas chargé. Donc il faut aller créer ta thread, en pensant bien à l'initialisation à charger tous les éléments dont t'as besoin. Ca c'était un peu galère.

-Et la solution pour ça ?

-La solution, c'est du test essai erreur. Et ça serait bien de voir ce qui se passe entre les threads. Parce que là c'est pareil pour la communication : tu fais une visualisation mentale qui est peu partageable quand tu travailles à plusieurs. Ce serait vraiment important d'avoir un outil qui te permet de tracer toutes ces interconnections, et ce serait bien aussi de pouvoir les commenter. Dire "là il se passe quelque chose, attention faut faire gaffe à ce niveau là, parce qu'ici justement moi je suis en train de modifier une interconnexion". Même pour la collaboration ce serait vraiment important.

-Et mentalement, qu'est-ce que tu t'étais représenté justement pour gérer ce problème ?

-Bah tu vois t'as ton programme, t'as ta colonne principale et t'as des colonnes parallèles qui sont tes threads. Et y a pas forcément de liaisons qui sont possibles à certains niveaux, la liaison se fait par en-dessous. Arriver à visualiser le point de liaison où tu peux faire communiquer, et bien percevoir que ce point il ne peut pas être plus haut. Ce qui est autorisé et interdit par ton langage aussi. C'est encore pire sur Android Studio, ils veulent être sûrs que tu ne vas pas leur bloquer l'application avec 20000 threads qui vont circuler à droite à gauche, ils veulent contrôler l'utilisation CPU. Mais les contraintes de programmation liées aux langages que tu utilises peuvent être intéressantes à représenter aussi: thread ou pas thread par exemple. Ou encore les cas où quand ce que tu écris n'est pas très correct par rapport au langage, même si ça marche. Et si on te le dit au fur et à mesure, tu vas être capable d'optimiser ton code. Si on te le dit après deux mois de développement, tu vas te lancer dans des tests, et puis en CPU tu vas tout exploser, ton truc il va mettre 25 secondes pour se charger, bah là l'optimisation va être super coûteuse.

-Et tu imagines quoi dans ce cas pour t'aider ?

-Bah un truc sur le mode de la suggestion qui te dirait "là t'as écrit ça", "tu aurais dû écrire ça", "attention là t'appelles deux fois ce modèle à deux endroits différents, c'est pas propre". Et d'ailleurs, si tu arrives à représenter les liens, ça tu peux l'analyser. "Je veux bien faire fonctionner ton code, mais c'est moche". Et ça graphiquement ça pourrait être indiqué. Après t'as des fichiers qui commencent à être…de la merdasse quoi, comme ma *global_view*. Ca fait 1300 lignes. Donc t'essayes d'organiser un petit peu, là où tu fais toutes tes connexions, avec tous tes signaux et on fait ça plutôt au début. Et puis après les définitions de tes classes particulières, quelques fonctions et après t'as ton *main*. Mais tu vois déjà là ce serait bien d'avoir une structuration qui est un peu contrainte. Tu

pourrais minimiser des parties, genre la partie des signaux, ça simplifie ton fichier, ce qui existe déjà pour les classes. Arriver au niveau du code à pouvoir simplifier ton fichier pour vraiment accéder à ce qui t'intéresses. Et en même temps, voir éventuellement que si je travaille sur cette partie là, je pourrais avoir sur la droite "tiens cette variable là, elle apparaît où, où elle est définie, utilisée, redéfinie?". Parce que là t'as des trucs oui, mais ça t'oblige à monter et redescendre dans ton fichier et te balader, c'est chiant.