How Live are Live Programming Systems?

Benchmarking the Response Times of Live Programming Environments

Patrick Rein[★] Stefan Lehmann[★] Toni Mattis[★] Robert Hirschfeld[★, ‡]
[★] Hasso Plattner Institute, University of Potsdam, Germany
[‡] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

The idea of live programming has been applied in various domains, including the exploration of simulations, general-purpose application development, and even live performance of music. As a result, different qualitative definitions of the term *live programming* exist. Often, these definitions refer to a sense of "directness" or "immediacy" regarding the responses of the system. However, most of them lack quantitative thresholds of this response time. Thus, we propose a survey of live programming environments to determine common response times the community regards as sufficient. In this paper, we discuss the design of an initial survey focusing on general-purpose live programming environments. We describe the selection process of systems and the benchmarking model to measure relevant time spans. We illustrate the potential outcomes of such a study with results from applying the benchmarking model to Squeak/Smalltalk and the Self environment. The results hint that a quick adaptation of the executable form might be a common feature of live programming environments.

CCS Concepts

•Software and its engineering → Integrated and visual development environments; •Human-centered computing → *Empirical studies in HCI;*

Keywords

live programming, system response time, benchmark model, empirical study

1. INTRODUCTION

Live programming has been available for a variety of domains, such as performance art, developing graphical user-interfaces, and also general application development. In all these domains, the impression of programming a live system is created through a variety of mechanisms, including visual representations of the system state, results of executions with example input, or direct and persistent manipulation of applications objects at runtime. Besides general factors such as how intuitive the visualization is or to which extend the runtime state can be manipulated, the delay between changes to the source code and observable changes in the system behavior is relevant to create an experience of "liveness".

One qualitative definition describes this experience as: "..., the computer wouldn't wait but would keep running the program, modifying the behavior as specified by the programmer as soon as changes were made." [24] This definition explicitly states that the change to the executable form should be applied "as soon as" the changes to the source were made and without the developer initiating it explicitly. Another qualitative definition mentions that the feedback on the system should be "...presented so as to be constantly present and constantly meaningful" [12]. Again the term "constantly meaningful" implies that there should be no noticeable period in which the feedback does not represent the runtime state.

These definitions can guide the development of features of a new live programming platform. However, it does not yet help programmers in judging whether the duration of the feedback loop is actually short enough to enable an experience of programming a live system. For example, two object-oriented live programming system with hot-swapping support might allow the exchange of methods during the execution of an application. However, in one system the translation and swapping of a method might take 10 seconds while in the other system the same process might only take 10 milliseconds. Both do comply with the qualitative definition but do greatly vary in the experience of liveness they can provide. Further, a short delay in updating a method in a running system provides the system designer with greater freedom in creating feedback mechanisms matching the application and domain at hand.

In order to provide guidance for future live programming platform development, we propose a benchmarking model for live programming systems to assess the duration of the feedback loop. The model separates individual phases to distinguish between system-dependent and application-dependent factors and is based on a notion of system-response times [21]. In a pilot study based on this model, we have also determined the duration of the feedback loop of two existing live programming systems: Squeak/Smalltalk [13] and Self [26]. These figures and the results of future studies might help platform developers with design decisions for new live programming systems.

In the following we will first describe the experience of live programming in more detail (section 2). Based on this general notion of live programming, we introduce our benchmarking model (section 3). We then describe the design of a study based on this model with the goal of analyzing various live programming systems (section 4). We also present first results from a pilot study using two exemplary systems (section 5). Finally, we discuss the preliminary results and potential outcomes from conducting an extensive study (section 6).

PX/16, July 18 2016, Rome, Italy
© 2016 ACM. ISBN 978-1-4503-4776-1/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2984380.2984381

2. DEFINING THE EXPERIENCE OF LIVE PROGRAMMING

Live programming environments are developed and used in many different domains. Most of them share the common goal of making programs easier to explore and understand. Some authors stated that live programming might be an adequate way to bridge the gap between the static source code and the dynamic behavior of an application [16, 20]. Some claim that live programming can enable this because short feedback loops might sustain the impression of causation between a change to source code and a corresponding change in the behavior of the application [25].

In general, these environments support live programming through a short time span between a change to source code and an observable change in behavior of the application. Additionally, they do apply the change while the application is running and without discarding the current state of the application completely. Thus, they preserve the context in which the change was applied [10].

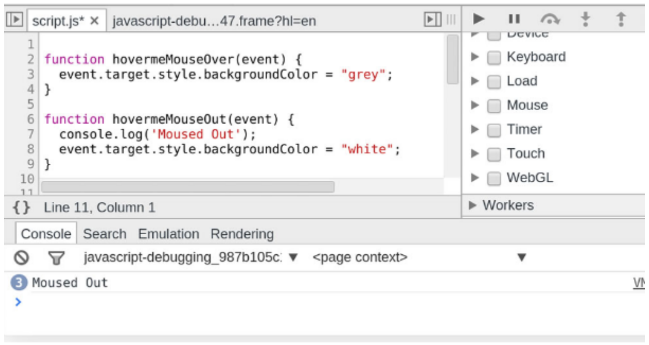
One definition of live programming uses these properties to qualitatively distinguish between different levels of live programming [24]. The levels 1 and 2 are concerned with the semantic representation of code an whether the representation is executable itself. Levels 3 and 4 are what current live programming environments typically support. Level 3 liveness is supported when an edit operation of unspecified granularity triggers any computation by the system. So, an environment in which unit tests are executed every time a file is saved, does support level 3 liveness. Level 4 liveness is supported when the changes to the program are applied as soon as the changes where made, without the user explicitly initiating the application of the change, and while the application is running or potentially so [23]. So the application would not wait until a user finishes a modification to run the application but the application constantly remains running.

Another definition of live programming, which is based on the concept of "steady frames", puts more emphasis on the preservation of context [12]. A steady frame is a way to provide live and continuous feedback which preserves the context of an activity. The goal is to make programming a continuous activity such as aiming with a water hose, instead of an activity with discrete independent steps such as aiming through shotting single arrows. An activity with a steady frame is organized such that "(i) relevant variables can be seen and/or manipulated at specific locations within the scene (the framing part), and (ii) these variables are defined and presented so as to be constantly present and constantly meaningful (the steady part)." [12] Therein, the selection of relevant variables mainly depends on the activity and the concrete task at hand. Further, the presentation of the variables also depends on the domain but should make relationships between the manipulated parts of the system and the goal easy to perceive.

2.1 Variations of Live Programming Environments

The different aspects of live programming are implemented on various levels, each with their own trade-offs. Each approach creates a different kind of live programming experience. Thus there is a spectrum of liveness which covers environments such as auto-testing setups, Read-Eval-Print loops (REPLs), and systems where tooling and application share the same runtime.

One group of live programming environments is based on data flow abstractions. They are used in particular domains, for example in live music production, visual signal processing, or graphical user interfaces (Viva [23], Max Language [7], Vivide [22]). In general, these environments represent continuous streams of data flowing through processing elements connected to each other. A



```
function hovermeMouseOver(event) {
  event.target.style.backgroundColor = "grey";
}

function hovermeMouseOut(event) {
  console.log('Moused Out');
  event.target.style.backgroundColor = "white";
}
```

```
3 Moused Out
```

Figure 1: A screenshot of the Chrome developer tools showing a modified event handler (hovermeMouseOut). The logging call was added to the handler and the console below shows the resulting console output. The new behavior became active without a reload of the page. [11].

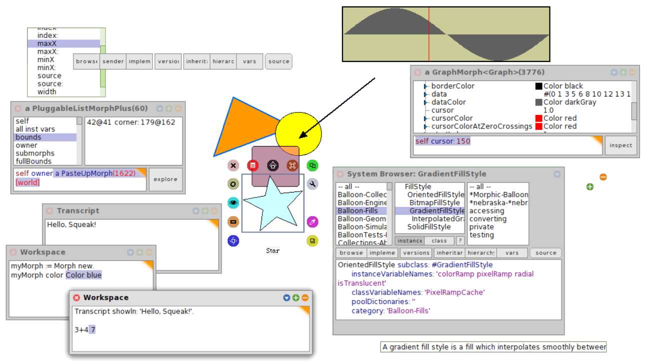

a PluggableListMorphPlus(60)
a GraphMorph<Graph>(3776)
Transcript
Hello, Squeak!
Workspace
myMorph := Morph new.
myMorph color: Color blue
Workspace
Transcript showln: 'Hello, Squeak!'.
3+4
Star
System Browser: GradientFillStyle
A gradient fill style is a fill which interpolates smoothly between

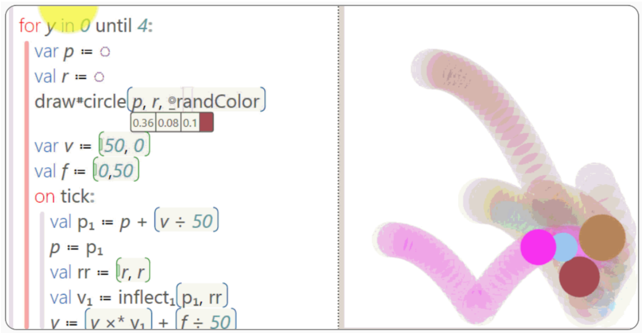
Figure 2: A screenshot of Squeak/Smalltalk showing graphical objects as well as tools for inspecting them in the same environment. [4].

modification of the behavior of single processing elements or of their combination changes the output immediately and thereby produces immediate feedback as if "one physically changed wires and parts on a video-processing circuit while it was plugged in and running." [23]

Data-flow-based environments can provide immediate feedback and preservation of context through the underlying abstraction. However, they are limited to domains which can be expressed with data-flow abstractions. In contrast, general-purpose environments have to introduce additional mechanisms to allow for live programming.

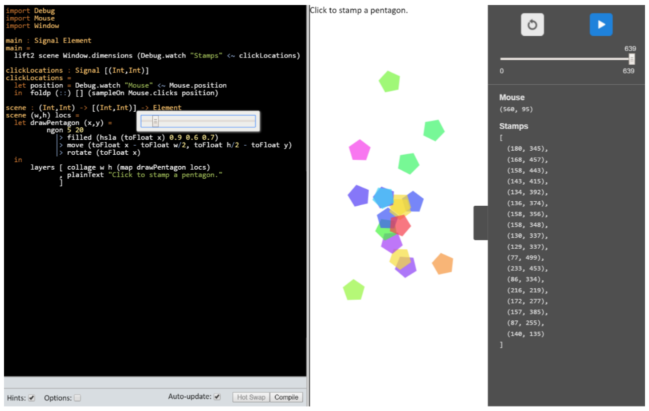
Some environments implement live programming through hot-swapping support. Hot-swapping allows to replace parts of the executable form of an application while the application is running. As a result, the application can continue the execution from the state at which the change was applied with changed behavior. Examples of hot-swapping-based live programming systems include various Smalltalk implementations and also the Chrome web development tools which allow for replacing graphical elements of web pages and also for replacing JavaScript functions without reloading a webpage [11]. The technique of hot-swapping in general allows for incremental and therefore quick changes to the executable form and preserves the current context of a change.

Other environments solve that problem by re-starting the application on a change. In order to bring the application back to the state at which the change was applied, the environment replays the



```
for y in 0 until 4:
  var p ≔ ○
  val r ≔ ○
  draw▪circle(p, r, ▫randColor)
                    0.36 0.08 0.1
  var v ≔ (50, 0)
  val f ≔ (0,50)
  on tick:
    val p₁ ≔ p + (v ÷ 50)
    p ≔ p₁
    val rr ≔ (r, r)
    val v₁ ≔ inflect₁(p₁, rr)
    v ≔ (v ×* v₁) + (f ÷ 50)
```

Figure 3: A screenshot of the APX system showing traces of the positions of the drawn circles. On a change to the code on the left, the traces would change and show what path the circles have taken with respect to the new code [18].



```
import Debug
import Mouse
import Window

main : Signal Element
main =
  lift2 scene Window.dimensions (Debug.watch "Stamps" <~ clickLocations)

clickLocations : Signal [(Int,Int)]
clickLocations =
  let position = Debug.watch "Mouse" <~ Mouse.position
  in  foldp (::) [] (sampleOn Mouse.clicks position)

scene : (Int,Int) -> [(Int,Int)] -> Element
scene (w,h) locs =
  let drawPentagon (x,y) =
          ngon 5 20
            |> filled (hsla (toFloat x) 0.9 0.6 0.7)
            |> move (toFloat x - toFloat w/2, toFloat h/2 - toFloat y)
            |> rotate (toFloat x)
  in
      layers [ collage w h (map drawPentagon locs)
             , plainText "Click to stamp a pentagon."
             ]
```

Click to stamp a pentagon.

```
Mouse
(560, 95)

Stamps
[
  (180, 345),
  (168, 457),
  (158, 443),
  (143, 415),
  (134, 392),
  (136, 374),
  (158, 356),
  (158, 348),
  (130, 337),
  (129, 337),
  (77, 499),
  (233, 453),
  (86, 334),
  (216, 219),
  (172, 277),
  (157, 385),
  (87, 255),
  (140, 135)
]
```

Figure 4: A screenshot of the Elm debugger showing the code on the left side, the current runtime state in the middle, and a tool for navigating back in time on the left. Again changing the code on the left changes the resulting state directly [2].

application. To achieve this they have previously recorded any input events and replay them at appropriate moments. The resulting state of the application is not the same as it was when the change was applied, but reflects how the state would look like if the change had always been there. Notable examples of such environments are Elm [8] or APX [18]. These two approaches to general-purpose live programming take different views on the past of the current state. The former environments treat the past as "immutable" while the later treat it as "mutable".

These kinds of environments enable level 4 liveness. Other environments do achieve level 3 liveness through a quick way to restart the system and to execute tests or small pieces of code. This way the state is not preserved but the developer can quickly get feedback on changed behavior of the system regarding the exemplary code segments.

3. LIVE PROGRAMMING BENCHMARK MODEL

There is no general implementation strategy for live programming systems as the applicable mechanisms depends on the domain at hand. Hence, we aimed to create a general model to allow for comparison between these different kinds of live programming systems.

The model is based on a general model for response times in user-interfaces [21]. This model measures the time from the moment users initiate an action to the moment they can recognize any kind of response from the system. This response does not have to be a final result corresponding to the action. It may also be a busy indicator making clear that the system accepted the input and is processing it.

While this response time model targets user interactions in general, our model focuses on the specific interaction of changing an application. We measure the response time as the time span between the completion of a change to the source code and an observable change in the behavior of the application (see Figure 5). Conceptually, this process comprises a modification of the program itself and not only of the mere source code.

In some environments, the overall duration of this time span does not depend solely on the capabilities of the platform but also on the application under development. A platform such as Squeak/Smalltalk can, for example, support live programming as it can exchange any part of an application while it is running, so if a programmer changes the behavior of a graphical object, the object will instantly behave differently. However, there will be no immediate feedback if, for example, a property of a graphical object depends on behavior only executed during the initialization of that object.

To distinguish between the capabilities of the platform and the factors introduced by the application, we separated the response time in our model into two phases (see Figure 5): adaptation and emergence.

3.1 Adaptation

The *adaptation* phase spans from finishing a change to the abstract representation of the application to an updated executable form of the application in memory. The abstract representation of the application can be for example source code, a graphical model, or tiles. The phase starts when a user finishes a change, for example by dropping a tile in a new position or saving an edited text file. Converting this representation to an executable form often includes some form of compilation. After this translation, the application needs to be updated in memory. Some platforms are capable of replacing only some parts of the application ("hot-swapping") other require a reload of the binary. The adaptation phase is completed as soon as an executable form of the updated behavior is loaded into memory and can potentially be executed right afterwards.

3.2 Emergence

The next phase comprises the *emergence* of an observable change in the behavior of the application from the adapted executable form. An observable change can be for example a changed textual output on the console, a different color of a graphical element, or a changed way of moving of a graphical element.

The nature of the change can have an influence on when it actually becomes observable. If a graphical element gradually and slowly changes its color from blue to green, the change in behavior might only be visible after the color is "different enough" for the observer to recognize the difference. The model does not cater for these cases and we assume a distinct change as soon as the new behavior is executed. In general, the observability of such changes can be improved by the overall design of the feedback mechanism. These factors are not captured by our model.

Further, the observation of changed behavior is independent of the feedback being correct in the sense that it behaves as expected. In contrast, the quick observation of behavior that does not seem to be meaningful becomes useful as a hint that there might be an error in the code.

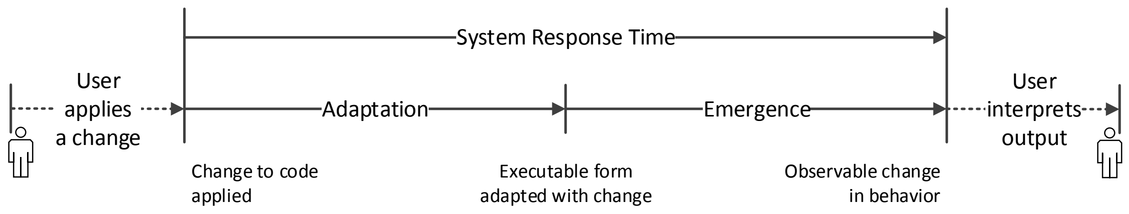


Figure 5: The phases from a modification of the code to an observable change in the behavior, which provides developers with feedback on their changes (adapted from [21]).

For a change to emerge from a modification of the source code, the updated behavior has to be executed. How quickly this occurs after the adaptation phase depends largely on the application. A game or a simulation which is continuously running might execute the new behavior in the next iteration. A graphical tool with a complex initialization phase might require a restart to execute the new behavior.

Nevertheless, this also depends on the capabilities of the platform. A data flow system does exclude this issue on a conceptual level, as there is a continuous flow of data and as a result it provides continuous updates of outputs. Further, platforms which allow for a mutable past, can rerun initialization phases. Other platforms might provide tooling to rerun small examples or tests continuously.

4. STUDY DESIGN

As a first application of the model, we plan a study on response times of general-purpose live programming environments. To ensure comparability between the results of individual environments, we devised criteria on how to select relevant parts of a system to benchmark using our model.

4.1 Benchmarking Process

We created a common benchmarking process as live programming environments have different units of change (for example functions, classes, objects, files, nodes, tiles). As we are interested in the liveness a developer can expect from an environment, we want to ensure that we are measuring the response time for changes as a developer would create them. For example, while the Elm environment uses a functional language, so most changes will affect functions, the unit of change is a file. A developer is not only starting the adaptation phase with a single function, but always with a complete file. The single steps to apply the model to an environment are:

1. Determine relevant units of change from the developer perspective. Use the most common ones.

2. Determine relevant operations on these units of change (add, modify, delete, compound operations (for example refactorings)).

3. Select representative code samples which reflect the complexity or length of a common unit of change of the environment. The sample should also work in combination with any emergence mechanisms of the environment, for example a replay system works well for a system with user inputs and does not match a long-running computation.

4. Apply the model from the perspective of a developer, so the run should include all activities which would be triggered when a developer saves a unit of change (for example regarding logging or persisting changes).

4.2 Candidate Environment Selection

In our first study, we want to focus on live general-purpose programming and learning environments. First of all, we exclude environments dedicated to particular domains such as audio processing, video processing, or cyber-physical systems as some of them can make assumptions not applicable to general programming such as data-flow semantics or asynchronous computation. Some of them also have very specific response time requirements as they control real-time processes. Our first study will include environments which satisfy the following four criteria without modifications to the environment.

(1) The environment has to allow the persistent development of a program. This means that there has to be a way to store and retrieve the resulting code and state of the application in the environment. This will exclude REPLs from this particular study (while REPLs do implement live programming features we want to keep the focus of the study focused on environments for persistent application development at this point). (2) From a user perspective the tools for editing the abstract representation of the application should run in the environment where the application is running. (3) The environment should allow level 3 live programming without any modifications to the execution environment or the standard library. (4) It has to allow for level 4 live programming without extensions for the execution environment. However, the environment can provide extensions for particular types of applications, for example graphical applications.

4.2.1 Candidates

Based on these criteria we have selected the following environments:

- Self [26]: General-purpose, prototype-based, UI manipulations can effect code manipulations, self-sustaining

- Lively Webwerkstatt [14]: General-purpose, prototype-based, web-based, self-sustaining

- Squeak/Smalltalk [13]: General-purpose, object-oriented, class-based, self-sustaining

- Etoys [9]: Learning environment, tile-based programming

- Elm [8]: General-purpose, functional, live programming capabilities introduced through tools

- SLIME [3] with CLISP [17, 1]: General-purpose, functional, self-sustaining

5. PILOT: SQUEAK / SMALLTALK AND SELF

To assess the nature of the results of the study, we conducted a pilot study with the Squeak/Smalltalk and the Self environment. Both environments enable level 4 liveness as they translate changes to source code incrementally to changes to the executable form. The executable form can also be updated while the application is running. They thereby implement an immutable past model, which makes the emergence time highly variable and almost completely dependent on the application at hand. Thus, we focused on the adaptation time in the pilot study.

5.1 Technical Specifications

All benchmarks were executed on the following system:

- Intel CPU i5-4690 @ 3.5 GHz, 4 Logical cores
- 7926 MB Main Memory
- Ubuntu 15.10

5.2 Squeak

In Squeak/Smalltalk, a permanent description of behavior is generally expressed through methods contained in classes. The relevant units of change are therefore individual methods, super class relations, and instance variable declarations. For all three elements, the developer can create, modify, and remove them. To keep the pilot study focused, we chose to only measure the most common activity, which is modifying a method.

5.2.1 Setup

To measure the adaptation phase duration from the perspective of a developer, we measured the duration of a call to the callback of the save button, respectively the keyboard shortcut for saving. This corresponds to the interactions of a developer during the ordinary development workflow as the user can only save one method at a time using the standard tool set. As a representative sample of source code we have selected all methods available to us in the Squeak/Smalltalk image at that point. These code samples include methods describing basic system operations such as sorting or filtering collections, as well as methods as parts of larger applications such as the code browser. This resulted in a test set of 50,525 methods. We measured each method 20 times.

The specifications of the execution environment are:

- Squeak 5.0 (Update 15113)
- Croquet Closure Cog [Spur] VM [CoInterpreterPrimitives VMMaker.oscog-eem.1405]

5.2.2 Results

An overview of all results (see Figure 6) shows that the measurements can be separated into two distinct groups, one above 750 ms and one below 50 ms. The group above 750 ms consists only of methods which describe an Etoys menu structure. These methods are treated differently by a compiler extension and thus result in such high measurements.

A detailed view on the group below 50 ms (see Figure 7) reveals that there is a slight increase of the adaptation phase duration with increasing method length, which is expected as longer methods should require more time to compile. Nevertheless, for all method lengths the measured times stay under 4 ms.

5.3 Self

In Self, the behavior of a system can be modified through objects or slots contained in objects. There are four types of slots: read-only, read-write, method, parent. Objects can be cloned and slots can be added, modified, or removed. Modifying read-only and method slots is only possible through the object editor, while the value of read-write slots can also be modified through evaluating expressions.

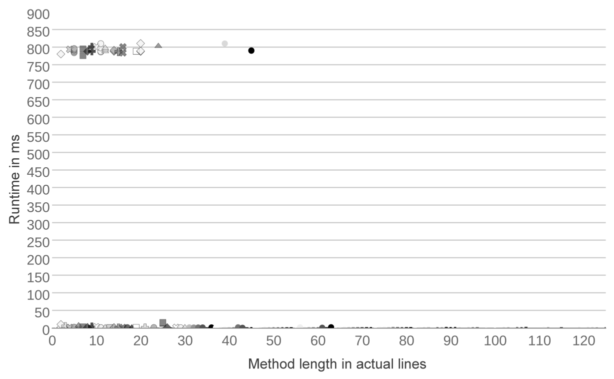


Figure 6: An overview box plot of the results of the Squeak benchmark showing the whole range of result values.

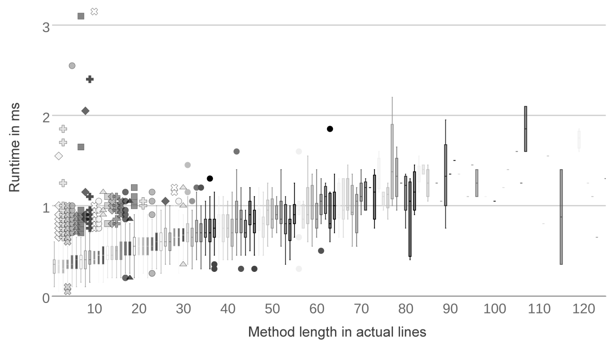


Figure 7: A detailed box plot of the major group of measurements from the Squeak benchmark.

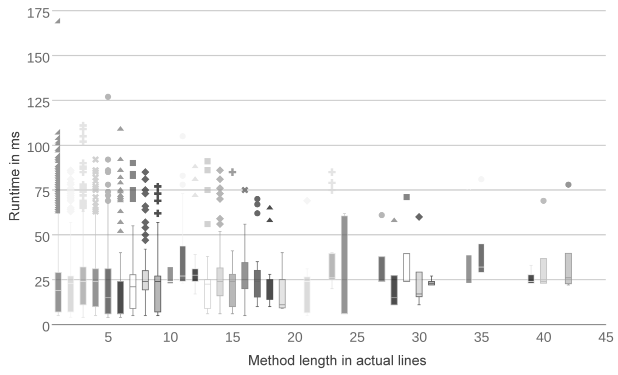


Figure 8: An overview box plot of the results of the Self benchmark showing the whole range of result values.

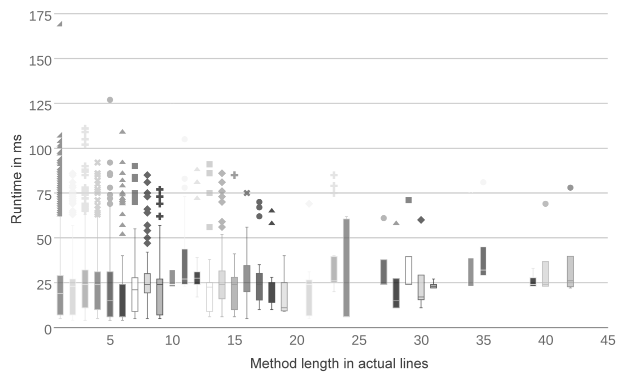


Figure 9: A detailed box plot of the major group of measurements from the Self benchmark.

Again, for the pilot study, we focused on the most common activity for changing the behavior, which is modifying a method slot. Similar to the Squeak/Smalltalk environment, programmers can only save one method slot at a time.

5.3.1 Setup

To measure the adaptation phase duration from the perspective of a developer, we have instrumented the method in the code editor object which is called when the user saves a function. In there, we have measured the duration of the function call which compiles and installs a new version of a function.

As a representative sample of Self source code we selected 13 core traits (string, list, collection, vector, clonable, integer, float, morph, rectangle, point, random, hashTableSet, outliner). Most of these traits contain basic system code and the outliner trait contains the more complex behavior of a graphical tool. We measured the duration of the adaptation phase for all their containing functions. This resulted in a candidate set of 729 functions. During the benchmark, we triggered the save callback function for every function 5 times. The specifications of the execution environment are:

- Self Virtual Machine 4.1.13
- World Builder Script from commit 978982b of https://github.com/russellallen/self

5.3.2 Results

An overview of the results (see Figure 8) shows several outliers beyond 200 ms. These are single measurements and do not stem from single method slot modifications. Thus, we attribute these to garbage collection or operating system scheduling artifacts.

The measurements below 200 ms (see Figure 9) show no obvious increase in the adaptation phase duration with an increasing number of lines of code. This hints that the compile time only makes up a small portion of the overall adaptation phase duration. Nevertheless, the median duration stays under 50 ms and except for 8 outliers all measurements stay under 200 ms.

6. DISCUSSION

The measurements from the pilot study hint some conclusions which might be secured by a larger study. First of all, a short adaptation phase duration might be a common property of live-programming environments. Second, the evaluation of the emergence phase might only yield useful results for mutable past and data flow environments as they implement mechanisms to deal with this phase. Additionally, it is yet to be determined whether the model should explicitly account for the restoration of state.

6.1 Short Translation Phase

Both benchmarks yield that the median adaptation phase takes less than 50 ms for common modifications. This short adaptation phase duration might be a common property of live programming environments, given that they aim to invoke an impression of immediacy for changing the executable form of the application. From the perspective of a designer of live programming environments, a short adaptation phase might thus be considered a necessary feature. However, the results from the pilot study are not reliable enough yet to state an adaptation phase duration upper boundary.

6.2 Emergence Phase

A consistently short emergence time is beneficial for a live programming environment, as it might enable constant feedback for developers. However, immutable past systems, as Squeak/Smalltalk and Self, do not guarantee a short emergence phase without extensions. Thus, our model might yield many "not applicable" results for the emergence phase duration for similar systems.

However, to still demonstrate common emergence times in these systems, it might be beneficial to analyze the system in combination with at least one extension which enables a consistently short emergence phase. For example, in Squeak/Smalltalk we could take the Morphic framework into consideration, which allows for the periodic execution of callbacks on graphical objects. We would then measure the emergence based on common examples for that extension. This however impedes comparability between systems, as each extension might have a different optimal example.

Another way of measuring the emergence phase in a comparable way might be to create a set of programming challenges which cater for different kinds of live programming systems. These should then be implemented in each system and the emergence of a specific behavior might be measured. By doing so the benchmark might also describe the design spectrum of live programming experiences.

Further, the emergence phase does not take the psychological effects of the feedback into account. It only measures the response time as a technical property of the system. We assume that the way the feedback is presented substantially influences the time it takes the programmer to gain insights from the changed behavior of the system. For example, if the results of executions on multiple example inputs are presented it might become easier for the programmer so grasp the overall behavior of the system.

6.3 Restoration of State

So far, both analyzed environments support an immutable past. For these kinds of environments, the restoration of state is not an issue, as they keep the state intact and only modify the executable parts of the system. However, systems with a mutable past do have dedicated mechanisms to restore the state which bring the application to the same point in time based on recorded input events. Currently, the model attributes this replaying of events into the emergence time. However, the model could also account for this phase separately as a third phase between the adaptation and the emergence phase. However, only these systems would require this separate phase as data flow and immutable past systems do not have this phase. Additionally, live programming systems with level 3 liveness also do by definition not require this phase and would also not yield useful results. Thus, we currently favor the two phase model in combination with the qualitative distinction between different kinds of experiences of live programming.

7. RELATED WORK

One work describes a formal model of live programming concerned with the validity of bidirectional edits between code defining a user interface and the rendered version [5]. Further, they describe a formal model for what we described as immutable past systems. However, they conducted no dedicated analysis of the response time of the proposed system. Based on this work another work also classifies live programming systems from a technical perspective into two categories. One class applies changes in real-time and does not change state from past executions and the other class records events and replays the complete execution on changes [19]. These categories correspond to our proposed categories of mutable past and immutable past systems.

Other work focused specifically on implementation strategies for level 4 liveness in declarative visual programming environments [6]. It also contains a theoretical analysis of the asymptotic complexity of the strategies and an empirical study of their runtimes for given examples. The empirical study also differentiates between different factors. However, the underlying model is designed for level 4 liveness visual programming environments, while our model is designed to allow for a variety of live programming environments.

General human-computer interaction studies also address the issue of quantitative thresholds for system response times. We have based our model on a general model for system response times [21]. This model was devised by surveying existing studies on the impact of the system response time on the productivity and well-being of software users. Another human-computer interaction guideline for the optimal system response time was derived from psychological cognitive studies on the processing capabilites of the human mind [15]. Two remarkable thresholds are the 100 ms threshold for preserving an impression of causality and the 1 s threshold after which users might wonder whether their command was actually accepted. Both guidelines could also be used by live programming system designers in decisions affecting the system response time.

8. CONCLUSION AND FUTURE WORK

Live programming has spread to various domains and there are qualitative definitions describing the experience of live programming. However, live programming environment designers have no guidance yet regarding the overall system response time they should strive for. We proposed a model of the system response time of live programming environments distinguishing between the adaptation of the executable form of a system and the emergence of observable changes in the behavior of the system. We described the design of a response time study of general-purpose live programming environments based on this model and gathered first results in a pilot study with Squeak/Smalltalk and Self. The observed median adaptation phase duration of under 50 ms hints, that a short adaptation phase might be common to live programming environments. Results from a broader study might result in a guideline for live programming system designer to judge whether their system is already "live" enough with regard to its response time.

8.1 Future Work

The pilot study hinted some conclusions regarding the characteristics of live programming environments. To gather more insights and strengthen the conclusions, we will conduct the planned study on the response times of general-purpose live programming environments. So far, the pilot study did not yield results for the emergence time. However, we hope to gather data on the emergence phase by defining a set of challenges with well-defined required observations.

To validate that the results of the model can characterize live programming benchmarks, it would be beneficial to apply the model to environments which are traditionally not seen as live programming environments. To gather measurements for the adaptation, as well as for the emergence phase, the environment would still need to supply some form of feedback. In order to illustrate how such systems might achieve level 3 liveness, an analysis of systems with automatic test execution might be of interest.

Another interesting dimension for an analysis of live programming environments would be implementation strategies, such as incremental compilation, check points, or dynamic method dispatch. If the impact of these on the adaptation and emergence phase could be determined, live programming system developers could chose appropriate strategies for their system at hand.

Live programming systems are always a combination of technical architecture, application domain and a particular task at hand. They implement liveness on several levels and provide different experiences of liveness (see section 2). A taxonomy of all kinds of live programming systems including domain-specific environments in combination with a quantitative evaluation as proposed in this paper, might result in a general overview of the variety of live programming systems. The combination of measured adaptation phases and implemented feedback mechanisms might also give insights in the required maximum adaptation phase durations for specific kinds of feedback.

Acknowledgments

We would like to thank all participants of the PX/16 writers workshop for their valuable feedback and criticism bringing forward the discussion on the characteristics of live programming systems. We also gratefully acknowledge the financial support of the Research School of the Hasso Plattner Institute and the Hasso Plattner Design Thinking Research Program.

9. REFERENCES

[1] CLisp GNU Man Page. Online at http://www.clisp.org/impnotes/clisp.html (accessed 14th of June 2016).

[2] Elm's Time Traveling Debugger. Online at http://debug.elm-lang.org/ (accessed 14th of June 2016).

[3] SLIME Project Page. Online at https://common-lisp.net/project/slime/#platform (accessed 14th of June 2016).

[4] Squeak Webpage. Online at http://www.squeak.org/ (accessed 14th of June 2016).

[5] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in UI programming. In *Proceedings of Programming Language Design and Implementation (PLDI) 2013*, pages 95–104, New York, NY, USA, 2013. ACM.

[6] M. M. Burnett, J. W. A. Jr., and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the Symposium on Visual Languages 1998*, pages 126–133, New York, NY, USA, 1998.

[7] Cycling74. The Max Language Documentation. Online at https://cycling74.com (accessed 14th of June 2016).

[8] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of Programming Language Design and Implementation (PLDI) 2013*, pages 411–422, New York, NY, USA, 2013. ACM.

[9] B. Freudenberg, Y. Ohshima, and S. Wallace. Etoys for one laptop per child. In *Creating, Connecting and Collaborating through Computing (C5) 2009*, pages 57–64. IEEE, 2009.

[10] R. P. Gabriel. personal communication, 2016.

[11] Google Inc. Chrome DevTools Documentation. Online at https://developer.chrome.com/devtools (accessed 14th of June 2016).

[12] C. M. Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, Sept. 2003.

[13] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM SIGPLAN Notices*, volume 32, pages 318–326. ACM, 1997.

[14] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel: A self-supporting system on a web page. In *Proceedings of the Workshop on Self-Sustaining Systems (S3) 2008*, pages 31–50. Springer, 2008.

[15] J. Johnson. *Designing with the Mind in Mind*. Morgan Kaufmann, San Francisco, CA, USA, 2nd edition, 2014.

[16] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the conference on Human factors in computing systems (CHI) 1995*, pages 480–486, New York, New York, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[17] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1965.

[18] S. McDirmid. A Live Programming Experience. Online at http://www.thestrangeloop.com/2015/a-live-programming-experience.htmls (accessed 14th of June 2016).

[19] S. McDirmid. Usable live programming. In *Proceedings of the Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2013*, pages 53–62, New York, NY, USA, 2013. ACM.

[20] D. A. Norman and S. W. Draper. *User Centered System Design*. Lawrence Erlbaum Associates, Inc., Publishers, 1986.

[21] B. Shneiderman, C. Plaisant, M. Cohen, and S. Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, Upper Saddle River, New Jersey, USA, international edition of 5th revised edition, 2009.

[22] M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of modification and use in data-driven tool development. In *Proceedings of Onward! 2014*, pages 185–200. ACM, 2014.

[23] S. L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Language Computation*, 1(2):127–139, 1990.

[24] S. L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of Workshop on Live Programming (LIVE) 2013*, pages 31–34. IEEE, 2013.

[25] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *CACM*, 40(4):38–43, 1997.

[26] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) 1987*, pages 227–242, New York, New York, USA, 1987. ACM.