DENICEK: Programming Substrate for Concrete, Collaborative, Interactive Programming

Tomas Petricek
tomas@tomasp.net
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic

Abstract

Research on interactive programming systems gave rise to a range of programming experiences, including programming by demonstration, local-first collaborative editing, structure editing, schema and code co-evolution, provenance tracking and output invalidation. Those experiences are compelling, but they are hard to implement on the basis of existing programming languages and systems.

We contribute the Denicek computational substrate. Denicek represents a program as a series of edits that construct or transform a document consisting of data and formulas. Denicek provides two primitive operations on series of edits, merging and conflict checking, that form the backbone of the implementation of the aforementioned programming experiences.

We discuss the architecture of Denicek, document notable design considerations and elaborate the implementation of the programming experiences listed above. To evaluate the proposed architecture, we use Denicek as the basis of a simple innovative data exploration environment. The case study shows that the Denicek computational substrate provides a pathway to the design of richer and more accessible interactive programming systems.

Keywords

Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:

Tomas Petricek. 2018. DENICEK: Programming Substrate for Concrete, Collaborative, Interactive Programming. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

intro

1.1 Programming Experiences

list

1.2 Substrate

how it works very roughly

1.3 Contributions

one main thing - substrate - with other things

2 Background

all the references

3 Walkthrough

conference organizer

1) PBD and interaction
2) Merging and schema evolution
3) Evaluation and maybe provenance

4 Architecture

also NAIVE REALISM

5 Implementation

(how each of the features is achieved)

1) PBD (create textbox but not button)
2) Merging (refactoring)
3) Interaction (add button)
4) Evaluation and invalidation
5) Schema evolution

6 Design Discussion

7 Case study

(data science environment)

8 Discussion

8.1 Heuristic evaluation

8.2 Limitations

maybe

DESIGN PROCESS = formative examples + evaluation case study

xx

xx

introduction
background
related work
design process / goals
case study
formative research
formative study
analysis
system
implementation
evaluation / heuristic evaluation
discussion and limitations

9 Introduction

The computational substrate using which software is built determines the capabilities that the software can provide. An imperative substrate that views programs as instructions modifying bytes in memory makes it almost impossible to allow end-user inspection or reprogramming of running software.

A computational substrate defines what software is built from. This may be objects as in Smalltalk, lists as in Lisp, or memory with data and code as in UNIX/C. The different substrates enable different kinds of programming experiences. For example, object-oriented programming has historically been linked to the development of graphical user interfaces (where objects can correspond to elements on the screen). It has also enabled the development of visual programming environments such as the Alternate Reality Kit, based on message sending between objects.

In principle, any computational substrate can be used to develop any programming experience, but the greater the impedance mismatch between the substrate and the desired experience, the more difficult it will be to provide the experience and combine it with the rest of the system and other programming experiences developed for the system. (One can implement support for programming-by-demonstration using C/C++, for example as part of a game scripting engine, but it will not work with the rest of the ordinary C/C++ ecosystem.)

9.1 Substrate

The question asked in this paper is, what would be the ideal programming substrate for supporting a range of programming experiences that make programs more collaborative, transparent and allows for a gradual transition from non-programmer to a programmer. We want a programming substrate that makes it easy to develop programming experiences such as:

- *Programming by demonstration* – Allow non-programmers to construct simple programs by performing examples of the expected behaviour. [11].
- *Local-first collaboration* – Multiple users should be able to use and modify a single program, preferrably without requiring a central server. [6]
- *Provenance tracking* – The execution of the program should leave an understandable trace that lets the user understand why program resulted in a particular result.
- *Schema evolution [extra-ish]* – When the user evolves the structure of the program, data and code should co-evolve automatically to match the new structure.
- *Notational freedom [extra-ish?]* – Allow users to adapt the program using a notation that suits them and is appropriate for the programming task at hand. [Joel]
- *Concrete programming [extra?]* – It should be possible to reuse parts of program or program logic without constructing abstractions, for example by managed copy & paste.[3, 4]

substrate as defined by [5]
[2, 11] [1]
[6, 7] [8–10] [12, 14] [13]

Joel's definition of substrate in Onward! Bret Victor talk https://www.youtube.com/watch?v=ef2jpjTEB5U

In what ways is a substrate "natural"?

thinglab - create line by cloning, it sticks to mouse pointer, clicking sticks it to something else squeak - has all the browsers (method search...)

computational substrate how it differs from computational media? more low-level - media suggests that there it comes

10 The whatever system

10.1 Document + Edits

defines

- selectors
- nodes
- edits

10.2 Walkthrough

* todo list? (or counter, but that is a bit boring)

11 Themes

* programming by demonstration - binding interactions to gui elements (event handlers) * provenance tracking - Amy Ko's whyline, Probe Log by HPI, enables linked visualizations * merging of edit histories / collaborative editing - bonus - can share restricted link to allow users fill out forms (allow partial edits only / def by selector?) * scehma change - change data & code accordingly * everything is an edit - interaction with the GUI - evaluation? tbd * copy & paste abstraction (requires finishing new approach to formulas!) - edit before copy to propagate edit to other places (or edit after copy to make it specific to a case) - higher order copying from https://tomasp.net/academic/papers/copy-paste/paint22.pdf * augmenters - cf. bonnie nardi (calls them something else - Jonathan says) - add programming by demonstration data wrangling gui to table (trigger interactions) cf. lorgnette

12 Applications

* todo list / counter / maybe too simple * (if used in the walkthrough, maybe something else? board game as in varv - tic tac toe? or 7guis?) * conference organizer * data exploration (ala histogram) * linked charts

13 Extras

* metablocks? * self-sustainability * some non-browser implementation of this (as in Varv?)

explicit structure self-sustainability notational freedom

Maybe have 'enabled' for edits afterall? (we can merge with conflicts and disable some edits, but keep them in history for info)

NOTES type Edit = Kind : EditKind Dependencies : Selectors list – only needed for evaluated edits

VALUE vs STRUCTURE distinction * good in theory, nice for implementation * tricky to use! needs some assistance tools

TODO - things to work on * "represent" edits somewhere in document as "library of functions" and then call those from buttons (rather than embedding them directly) allow some kind of abstraction (as in Histogram) to make them reusable * figure out how to do evaluation better (based on the stored abstractions? but need to store provenance...)

SEMANTIC CONDITIONS https://www.youtube.com/watch?v=NBnc2ToS_j0 (has a section on this in background)

SUBSTRATE DESIGN PROBLEMS * selectors - all for structure / index for data (but it is useful to allow others...) (multiselect also bad for checks!) * groups/conditions/preconditions (c.f. email to jonathan) tried conditions on edits; trying groups with check edits * what to do with "disabled edits"? for example when we remove all checked (before, this created edit groups with "check" but if the check was false, the group was ignored and this messed up merging - because we wouldn't know if the edit had any effect or not)

Evaluation * evaluated edits have to be migrated to the end (if there are conflicts, they are dropped) Think of this as maintaining a tree:

e3 | e2 evaluated | / e1 | e0

this has to be serialized as e0 -> e1 -> e2 -> e3 -> evaluated

evaluated edits do not became part of the main history but hang on the side

ISSUES * if we merge a thing with saved-interactions with something, hashes will change!

NOTES * ListAppendFrom - we need this, because we cannot encode this. * for records, we can RecordAdd(sel, fld, ..) @ Copy(sel @ [Field fld], src) but this does not work for lists - because we do not know the index! (and we cannot look into current document, because it will differ for saved-interactions)

TODO * many things with <tag> selectors currently do not work (e.g. 'matches' for highlighting) because if we collect path of a current node, we collect indices and get /some/2/another - and cannot tell if this matches /some/<li>/another - we'd have to collect more detailed path info!

INTERACTION * replay stored event handlers against the old version? (this way, adding an item to a speakers list gets migrated & adds a new table row!) * simlarly!! we need merge in order to apply edits to multiple targets (when you remove all items in a list, the indices change) (but I guess we should do this against version at the time of saving too....)

[this & evaluation = the unreasonable effectiveness of merging]

Notes on storing and reusing edits * references need to be represented as references so that they get updated (NO! not if we reply them against old version, which seems better - but there are 2 design choices) * how to apply them to multiple targets? use Move to update the selectors instead of replacing the prefix manually

IDEA: Type check edit groups to ensure they preserve structure but not individual edits eg when adding list item

CONDITIONALS https://toby.li/files/p311-radensky.pdf

REMAINING IMPLEMENTATION TODOs:

* Some kind of provenance visualization * Some kind of matchers/transformers mechanism (ideally to add interactive buttons to tables) * Apply to all (remove completed in TODO)

References

[1] Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. MIWA: mixed-initiative web automation for better user control and confidence. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software Technology, UIST 2023, San Francisco, CA, USA*, pages 75:1–75:15. ACM, 2023.

[2] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[3] Jonathan Edwards. First class copy & paste. Technical Report MIT-CSAIL-TR-2006-037, Massachusetts Institute of Technology, 2006.

[4] Jonathan Edwards and Tomas Petricek. Interaction vs. abstraction: Managed copy and paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*, pages 11–19, 2022.

[5] Joel Jakubovic and Tomas Petricek. Ascending the ladder to self-sustainability: Achieving open evolution in an interactive graphical system. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 240–258, 2022.

[6] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

[7] Clemens Nylandsted Klokmose, James R Eagan, and Peter van Hardenberg. Mywebstrates: Webstrates as local-first software. In *UIST'24: Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2024.

[8] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[9] Amy J Ko and Brad A Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578, 2009.

[10] Eva Krebs, Patrick Rein, Joana Bergsiek, Lina Urban, and Robert Hirschfeld. Probe log: Visualizing the control flow of babylonian programming. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*, pages 61–67, 2023.

[11] Germán Leiva, Jens Emil Grønbæk, Clemens Nylandsted Klokmose, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. Rapido: Prototyping interactive ar experiences through programming by demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 626–637, 2021.

[12] Roly Perera, Umut A Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 365–376, 2012.

[13] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. Linked visualisations via galois dependencies. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[14] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative functional programs that explain their work. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.