Multiagent Live Programming Systems: Models and Prospects for Critical Applications

Steven L. Tanimoto
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle 98195, Washington, USA
tanimoto@uw.edu

ABSTRACT

Live programming constitutes a human-computer symbiosis in which a human creative activity and a continuous computer execution influence each other. Usually, there is a medium of expression called "code" that the human(s) use to express desired behavior on the part of the computer, and the computer provides its feedback in the form of textual, graphical, audio, or other output. The most popular domain for live programming has been music synthesis ("live coding"), but the key features of live programming suggest it can play an important role in other applications, even process control or emergency management.

This paper breaks down live programming systems in terms of agents, both human and computational, their roles, and representations they typically create and act upon. It then comments on how multi-agent live programming systems could add new flexibility to information systems such as those that manage critical infrastructure or emergency response activity, such as during a COVID-19 type of pandemic or after a major earthquake.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Collaboration in software development**;

KEYWORDS

Software engineering, live programming, hot-swapping, models of live programming systems, multi-agent systems, agile development, coronavirus, COVID-19, earthquake, emergency management, safety

ACM Reference Format:
Steven L. Tanimoto. 2020. Multiagent Live Programming Systems: Models and Prospects for Critical Applications. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3397537.3397556

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397556

1 INTRODUCTION

1.1 Overview

This paper examines the architecture of live programming systems in terms of their component agents, data representations, and safety of executions. The purpose of this analysis is to help establish reference points for the research and development community for the design and analysis of new systems that offer alternative approaches to amplifying human creativity and productivity through computation. Possible application scenarios include "critical" ones such as computer-assisted management of responses to unfolding emergencies including pandemics and earthquakes, as well as "creative" ones such as the already popular application domain of music synthesis[Aaron and Blackwell 2013].

1.2 Motivation

The design of programming environments has a major impact on the experience of computer programmers[Edwards et al. 2019], including not only their productivity in terms of lines of code per unit time, and the quality of that code, but also the affective aspects of programming, and the programmers' creativity in the broader context of seeing the current task in the context of the entire project or series of projects.

Furthermore, by supporting "live" programming, an environment can enable a possibly more interactive programming experience that can enhance either the process or the end result of a programming session[Victor 2012][McDirmid 2013][Church 2017].

However, there are other domains where flexibility to quickly add new program elements to a software system could conceivably be extremely helpful. Such a domain is emergency management, when time pressures and unexpected software and information needs can arise. The time element is critical in pandemic response, earthquake response, wildfire control, and many other types of emergencies.

This paper is concerned with the question of how to design new systems that support live programming. It focuses on models that can serve as high-level design schemata, in the early stages of such a project, but where such a project may be to extend the effectiveness of an existing software application.

Why do we need models? One reason is that there are many ways to achieve the goals of liveness, and it is important to be able to readily analyze designs for possible systems with regard to a particular application. Another is that models help us to understand the space of possible live programming environments, to support an evaluation of their applicability to emergency management systems.

1.3 Benefits of Liveness

When either choosing a development environment or designing a scriptable software system, an important question is what the benefits of liveness might be. There are two main benefits.

immediacy of feedback to programmers: The possibility of low-latency responses from a running system, in terms of the semantics of the program being modified, has been the foremost attraction of live programming.[Kato 2017]. This is not only a convenience for debugging, but perhaps a key notion in a reinforcement learning situation in which a programmer is discovering how best to structure or tune a section of code. Having lots of trial/response interactions, as in big-data machine learning, makes a big difference in the results.

keeping continuity of execution: This is important in certain applications. Examples include musical performance, and as suggested later in this paper, emergency management or possibly the control of a nuclear reactor.
When something is wrong with the control program, we may not have the luxury of restarting from the beginning, but need to make a patch that can fix the problem without interrupting (or minimally interrupting) the execution. For example, a computation which expends resources (such as raw materials in manufacturing) to get up to a given state, cannot simply be restarted if that state is not perfect.

In order to achieve these benefits, liveness must be thoughtfully integrated into the programming environment.

1.4 Prior Work on Live Programming

Creation of software can be accomplished using many methodologies. Most involve traditional edit-compile-run cycles. But some use live programming environments, or possibly even non-coding interaction[Petricek 2019]. System support features include file management (naming, versioning, paths on the file system), editing assistance (refactoring, auto-completion, etc.), and runtime safety (type systems in programming languages, assertion mechanisms, unit-testing affordances, etc.)

Live programming environments supplement conventional features with forms of liveness[Edwards et al. 2019]. Liveness can sometimes lead to faster semantic feedback to a programmer writing program code. This may help the human express ideas at a rate closer to the rate of thought, rather than at a slower, belabored rate caused the encumbrance of more required actions on the part of the programmer. Liveness can be incorporated in a variety of styles within an environment, from full programming to merely parameter tuning[Kato and Goto 2016].

For more literature related to liveness, see the references in some recent works[Petricek 2019][Kato 2017].

2 BASIC MODEL COMPONENTS

The focus of this paper is on better understanding the design space of live programming environments that could be built in order to better exploit emerging tool technologies, artificial intelligence, and collaborative programming methodologies, and in order to broaden the scope of applications that can benefit from live programming. This section addresses the question of what sorts of agents should be considered in such designs, as well as what types of program and meta-program representations may be needed in the coordination of live programming processes.

2.1 Human Agents

The most important agents in a programming environment are the humans who work to create, shape, analyse, and document the programs. We can consider all of these activities to be roles of "programmers" broadly construed. In some possible systems, we may label these people with more specific terms.

Requirements Analysts: Those who develop specifications for software.

Coders: Those who design or write program code.

Evaluators: Those whose main function is to provide services to programmers that complement the creation of the program code. The roles of evaluators include: testers, judges, and arguably, documenters.

In addition to filling these roles, the human agents communicate with one another, and they are generally members of a team.

2.2 Computational Agents

The programmers work in a software-tool environment that we can conceptualize as a collection of computational agents that embody processes that perform services. Whether the agents are implemented with separate execution threads, or even separate hardware is an implementation issue that is somewhat orthogonal to the design of the agents' functionality. (We are concerned in this paper with the functionality, not the implementation.)

Some of these agents perform traditional IDE tasks such as offer and accept code completions. Others correspond to services that are not standard today but that will be needed to better exploit artificial intelligence and other technologies. Agents that perform small analysis or code modification tasks we might call "script elves," while agents that tackle larger tasks such as inferring programmer goals or finding nearest matching code on the web may be called "script wizards." Both types of agents observe the program code buffer. Those which modify code proactively (e.g., auto-correcting indentation) can be distinguished from those which only suggest modifications which must be accepted by the programmer before they are performed. The nature of these analyses and transformations may be either proactive or suggestion-only, according to programmer preferences.

In a traditional IDE such as the bare Java-tools configuration of Eclipse, the functions performed by these agents are considered services. By considering them to be agents, we reinforce the notion that these functions can be applied proactively to increase liveness.

Code modifiers: Script elves that handle simple changes to the code buffer. These include prettifiers, annotators, syntax highlighter, code completion acceptors, and some refactorers.

Code analyzers: Script elves that handle code completion suggestion, lexers, parsers, ancillary diagram (e.g., type graph) constructors, and detectors of conflicts between different programmers' versions.

Segmenter: In a live programming environment, parts of a source code buffer may contain valid program code, while other parts contain comments, free textual notes, or invalid program code. The job of a segmenter is a partition the text in

the buffer according to these classifications, with a primary goal of inferring the programmer's intent and enabling the execution of whatever program can be discerned as a result of this process.

The "Interpreter": If not broken into separate services, one agent might handle the whole of lexical analysis, parsing, and execution.

Hot-swapping agents: These handle live modifications to the machine code while it is being executed. They must ensure integrity of the running process and trajectory of program states, insofar as practicable.

Continuity agents: These may operate directly on program states to compensate for an "inconsistent history." For example, a live change to the dimensions of an array from 100 to 10 could suddenly cause an array index of 79 to be outside the legal range. The continuity agent might use a policy to change its value to 9, or perhaps to 6, to retain consistency in relative location.

Adjudication agents: When human agents or other agents come up with conflicting requests/requirements, an adjudication agent can assist with an analysis and optionally a resolution.

Basic execution agent: execution of the main process being designed or modified.

Approximate computing agents: for early feedback when the main computation takes too long to feel live. In graphics/visualization applications, for example, progressive refinement of displays can help reduce the latency between programmer action and programmer comprehension of the consequences of that action.

Speculative executions: These agents run variations of the main execution in anticipation of relatively likely possible changes to the program. These not only allow tactically-predictive liveness ("level 5"[Tanimoto 2013]) but can reduce the need for approximate computation agents by getting to future execution states early.

Simulation agents: When a computation involves not only machine calculation but control of physical and social systems, an execution is not a "throwaway commodity." Yet some forms of liveness (e.g., level 5) require multiple threads of computation exploring possible futures of the main thread. In an environment with physical/social consequences, these exploratory threads should typically be simulations to avoid the high costs of consuming physical resources or social capital.

Meta-execution agents: These include debugging assistants, profilers, and execution critics.

IDE Control: One particular meta-agent is responsible for coordinating all the other agents in the integrated development environment. It may use a set of rules and preferences to do this. For example, it may allocate speculative execution agents on the basis of estimated opportunity/cost.

2.3 Representations

Source code, while perhaps the primary representation of a computer program in a live programming environment, is not necessarily the only representation of concern. In general, representations in a live programming system are the information passed back and forth or pointed to by the agents. Important characteristics of relevant representations are discussed in the following.

Inscriptions: – human-intelligible representations, including program code, approximate program code, slightly transformed program code, comments, free text, drawing, formulas, intelligible records of gestures, and clear examples of input-output pairs (for example-based programming).

Machine code: – representations that are normally unintelligible to humans, but meaningful to a computer, such as byte code, machine language code, etc. Typically, the machine code is organized to facilitate hot swapping, e.g, as class definition byte code.

Structured representations: Parse trees, graphs, CASE frames, flow charts, type hierarchy diagrams, dependency graphs.

2.4 Control Schemes

Coordination of the agents and representations in a live programming environment can itself be handled with agents and representations specific to this task. However, we give some special attention to this aspect of the system here.

Rules, graphs and other representations that describe how and when representations are processed. For example, a control scheme could poll for changes in a pair of human-edited buffers, where two programmers are writing function definitions for a single execution thread. When updates are detected by the polling, inconsistencies between the two programmers' buffers will be resolved by an adjudication agent, and the results passed to a code swapping and continuity-of-execution agent.

Means for maintaining the experience of liveness may be provided for in the control scheme. For example, approximate computations may be run, and or certain computations may be automatically checkpointed for transparency on the basis of elapsed time.

3 MODELS

In this section we consider views of possible live programming environments in terms of their component agents. This aspect is a key one for a model of such a system. The model should incorporate other aspects, such as information and control flows, and representations, but here we focus mainly on the agents.

Figure 1 shows a model for a minimal live programming system, with one human programmer and a computer agent that interprets the code produced by the human.

Figure 2 shows a collection of agents in a hypothetical, general live-programming system. The agents are grouped into four categories beginning with the human live programmers and ending with meta-execution agents. One could also imagine interconnecting these agents with information flow paths, but due to the high degree of potential interconnection, the diagram would likely be unnecessarily messy.

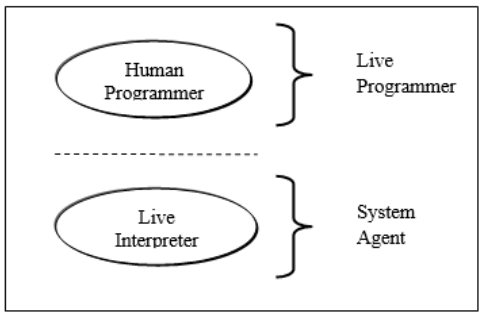

Figure 1: Model of the agents in a minimal system for live programming.

Note that data visualization tools are often valuable when the design of software involves making sense of large quantities of data. Agents supporting visualizations could be added to the collection in Fig. 3; however, they could also be considered to be part of a library of software tools, separate from the IDE, that can be linked into the software application under development.

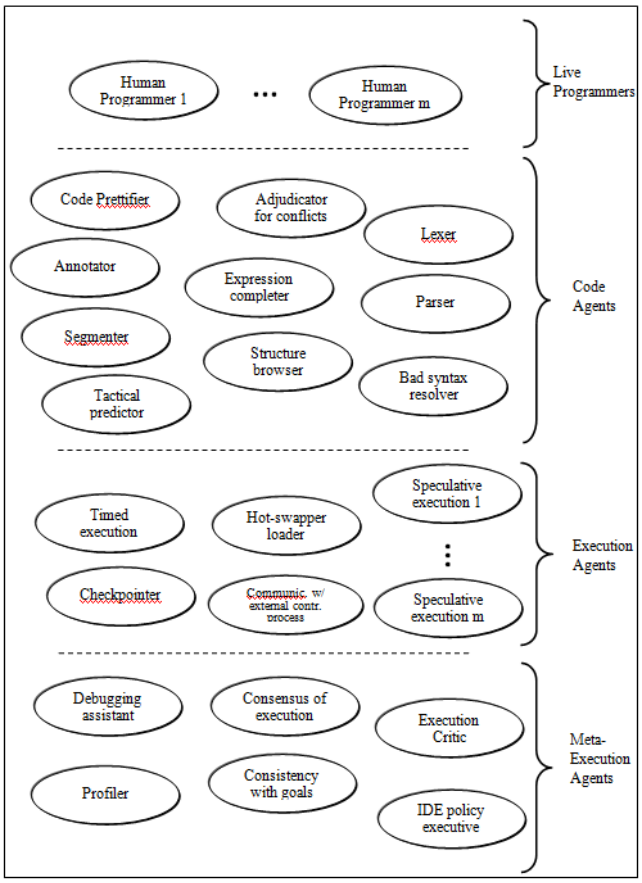

Figure 2: The agents in a general multi-agent model for a live programming system.

4 IMPORTANT ISSUES

4.1 Safety

Live programming has traditionally been promoted in a spirit of empowering the programmer, and making execution follow the programmer's latest actions. Although this may seem to violate safe practices such as careful vetting of code before execution, a later section of this paper discusses how live programming can be made safe.

4.2 Hot Swapping and Checkpointing

Maintaining continuous execution while reflecting the semantics (to the extent practicable) of the current version of a changing program poses challenges.

Hot swapping is the method of replacing a chunk of machine code in a running program by another chunk. Some live programming environments require this for continuity of execution of the main process. A somewhat analogous technique is "double buffering" in computer graphics, and this suggests one means of hot swapping: whenever a new method definition becomes available, redirect calls to the old method to go to the new method.

Certain limitations may sometimes be necessary to make hot swapping safe, such as having the main thread temporarily block if it reaches the point in the code where the hot-swapping is taking place.

Checkpointing is a means of saving past execution states in part or in total, for the purpose of backing up an execution, if required, to help quickly obtain a new modified execution that is semantically more consistent with the current version of the changing program. Making an execution as continuous as possible, and as consistent with a changing program as possible, is an open research problem. In live-coding practice for musical improvisation, the semantics of the execution are implicitly defined to be "whatever you get" (i.e., whatever the listener hears).

4.3 Coordination of Programmers

Live programmers may create conflicting function definitions or other inconsistent code. How such inconsistencies are handled can be considered at the system design level or at the level of control policies within the environment. One policy would be to resolve conflicts according to a hierarchy among the programmers; the lead programmer's version of a code block supercedes conflicting versions by other programmers.

4.4 Scripting vs. Live Programming

Scripting is typically considered to be a form of programming in which pre-written applications or libraries are called in order to coordinate execution flows that are primarily handled by those components. Live programming, on the other hand, is programming at any level of abstraction in an environment that offers the affordances of automatic transfer of code changes to the execution, etc. These two types of programming can be combined for live scripting, with no inherent inconsistency.

5 APPLICATION TO EMERGENCY MANAGEMENT SYSTEMS

5.1 Needs of Emergency Management Systems

An emergency management system (EMS) is a computer-based information system that keeps track of needs, resources, and responses during an emergency[World-Health-Organization 2013].

The unpredictable nature of each specific emergency has parallels in business, in which the rapid pace of business and technology changes require rapid adaptation of software products and plans. Agile development has been the most popular approach to software engineering in the past two decades under these conditions[Alaa and Fitzgerald 2013].

An EMS is typically structured either as a decision-support system[Turoff 2002] or a general-purpose digital communications platform. Since many emergencies occur over spatial regions, it is common to use maps or geographic information systems as a fundamental components of EMSs. While some emergencies such as a mine collapse, are focused in one location, broader emergencies such as earthquakes and pandemics require tracking multiple issues in a variety of locations. Keeping track of the issues and response plan status is part of what the EMS does. Interfaces for human response managers require affordances for quickly bringing up the status of a response effort and seeing where on the map it is taking place.

One of the most widely used systems is Virtual OSOCSS, developed under the auspices of the United Nations Office for the Coordination of Humanitarian Affairs, and as part of the Global Disaster Alert and Coordination System (GDACS). This system is intended to help deal with new emergencies anywhere in the world during their first few days (e.g., 72 hours). The primary types of emergencies addressed are earthquakes, tropical storms, volcanoes, floods, and droughts. In spite of its wide usage, it has been criticized for being oriented towards UN elites rather than the local teams on the ground who deal directly with the consequences of a disaster. Nonetheless, its development since 1999 has been studied, and recurrent patterns of new information-processing needs in it have been identified[Bjerge et al. 2016]. In fact, a major reason for Virtual OSOCSS not being more widely used by responders is that it still often fails to provide relevant information in a way that those in need of it can find it in a timely and appropriate fashion. Contributing to this problem is the fact that a plethora of new information formats continues to grow as more organizations in more countries either offer information or request information. Another issue is the insufficient internationalization of the interface; although Virtual OSOCSS supports more than one language in its static interface (English, French and Spanish), many of the responder teams in the world are not able to use these languages. The study by Bjerge et al found that over time, features were added to Virtual OSOCSS that improved flexibility of information access, and as that happened, the number of users increased.

The WebEOC system[Juvare-Inc. 2020] is a software product of Juvare, Inc., that is licensed to many local government agencies in the United States. It uses the Internet and browsers to give users access to EMS functionality. The company offers a short course on scripting with HTML and JavaScript for the purpose of creating new front-end "boards." However, back-end functionality is apparently not scriptable by end-users.

If a new feature is needed during an emergency, such as a means to integrate a new type of data, agile methodology has typically been the fastest way to incorporate new functionality. However, even that relatively flexible means to get the new code up and running may require too much time.

In a viral outbreak or pandemic such as with COVID-19, there may be a series of connected emergencies over a period of months. As a scenario in which live programming could help, consider the following. During the COVID-19 pandemic, there was a shortage of ventilators in many cities.

> A manufacturer offers to produce N ventilators of a slightly new type, B, which have certain limitations in capacity due to the unavailability of the exactly correct part C. The quantity for this proposed device cannot be integrated (within the EMS) with the expected ventilator supply predictions due to restrictions built into the EMS's ventilator specification data format. An immediate change is needed to the software to permit the type-B ventilators to be included. Due to this incompatibility, the human emergency coordinator, under time pressure and other stress, refuses to entertain the prospect of the type-B ventilators, which nonetheless could save lives.

Although this example is somewhat hypothetical, incompatibilities between software systems and data formats have long been a challenge for complex information systems. When time is of the essence, the ability to make changes promptly becomes urgent.

5.2 Incorporating Live Programming

Disaster response management often suffers from inadequate sharing of information during the response[Bharosa et al. 2009], which is an issue orthogonal to centralized decision-making.

The unpredictable nature of many types of emergencies means that the software systems that help manage responses to emergencies[Neville et al. 2016][Scholl 2019] should be easily adaptable, should their affordances not adequately match the needs of a new situation. While agile methodologies may help to allow last-minute software modifications to work[Alaa and Fitzgerald 2013], the live-programming methodology offers some complementary features that might be helpful.

Human factors are an important dimension of emergency response system[Park et al. 2014]. Liveness in the programming environment can help in making the system and its programmability easier to learn by a programmer who might not have been part of the design team for the system or who has not been fully trained in the system details.

Making a complex information system live-scriptable is technically feasible; it is well exemplified by the Analyst system[Thomas 1997][Xerox-PARC 1987], implemented in SmallTalk at Xerox PARC, before the existence of Microsoft Office for Windows.

5.3 Training Live Programmers

The live programming component of a critical application system affords the human programmer great power to quickly make changes to the system. Such a programmer must be well-educated in the domain, and trained to be able to use these affordances effectively and safely.

Let's call a live programmer for an EMS an EMLP. An EMLP must be familiar with the field of emergency management, with the structure of the EMS and with programming in the system's language. An EMLP should be a member of the emergency response team, and should be ready to communicate with team members if needed as part of evaluating the latest software needs and their impacts. In order to be ready to assist in an emergency, the EMLP should already be practices in the live-programming environment, and should have participated in drills which simulate actual emergencies.

5.4 Safety Concerns

Further to the issue of safety mentioned earlier, in an EMS or critical infrastructure, it is fundamental to minimize the risks of system downtime, accidental catastrophic command/policy mistakes or grossly incorrect parameter values, interference with proper operation, reduced transparency, or sabotage. Mitigation and prevention of these types of failures can be designed into an EMS without eliminating live programming. Consider, for example, minimizing risk of accidental catastrophic action by a live coder. First, a live programming component of a large system must be understood as one component, and not the entirety of that system. That component can be all or partially sandboxed, depending on the circumstances.

Simulation in a sandbox is one approach. Simulation is currently an important methodology in understanding how the dynamics of emergency management plays out[Gonzalez et al. 2016]. Live programming affordances should be connected to the simulation facilities.

In an emergency management system, live programmers should debug new functionality with embedded simulations that can use the real data as input, but can only provisionally alter the state of controls of the main system. Because the provisional feedback is live it achieves one of the two aims of liveness: rapid feedback to the programmer.

A suitably debugged and vetted new software component can then be hot-swapped into a main execution with reduced risk.

Another approach to reducing the risk that live programmer might cause damage is to incorporate a detailed risk-analysis process into the basic computational fabric of the system. Like systems that track bounds on numerical error in scientific computations, a risk management subsystem could make inferences about types and levels of risk associated with changes to the program code.

Other safety hazards such as sabotage, are not necessarily worse because of live programming. Some, such as the prospect for reduced transparency, may be better, because live programming environments are typically designed in a matter to expose the functionality of API calls and expose more details of the state of a system than usual.

5.5 Other Critical Infrastructure Systems

While natural disasters are the main targets of systems such as Virtual OSOCSS, information systems that control power, transportation, water, or communications networks are also in need of flexible and safe controls. It is an open question whether live programming facilities can offer something valuable that enhances the ability of such systems to respond effectively to unusual circumstances. For example, one can speculate that the Fukushima nuclear reactors might have behaved differently if either the control system had been more flexible, or if its simulation facilities had been so flexible and powerful that live programmers working with it would have more easily understood the risks the reactor posed in the face of possible tsunamis.

6 CONCLUSION

Adding a live-programming component to an existing software systems can be a means to more flexible and timely responses to changing software needs. While there are challenges in making a symbiosis between a live programming environment and a facility such as an emergency managements system work well, attempts to do it appear justified by the possibility of doing a better job in the ultimate management of emergencies or critical infrastructure.

ACKNOWLEDGMENTS

Thanks to Jochen Scholl, Burr Stewart, the anonymous reviewers and the PX/20 organizers for their discussions and/or encouragement regarding this paper. Thanks go also to the participants of the online lightning-talks session of PX/20 held on March 27.

REFERENCES

Samuel Aaron and Alan Blackwell. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages *(FARM '13)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/2505341.2505346

Ghada Alaa and Guy Fitzgerald. 2013. Re-Conceptualizing Agile Information Systems Development using Complex Adaptive Systems Theory. *Emergence: Complexity & Organization* 15 (September 2013), 1–23. Issue 3.

N Bharosa, J. Lee, and M. Janssen. 2009. Challenges and obstacles in sharing and coordinating information during multi-agency disaster response: Propositions from field exercises. *Information Systems Frontiers* 12 (2009), 49–65.

Benedikte Bjerge, Nathan Clark, Peter Fisker, and Emmanuel Raju. 2016. Technology and Information Sharing in Disaster Relief. *PLoS One* 11 (2016). Issue 9.

Luke Church. 2017. Becoming Alive, Growing up. Invited keynote, LIVE 2017, workshop at SPLASH/OOPSLA.

Jonathan Edwards, Stephen Kell, Tomas Petricek, and Luke Church. 2019. Evaluating programming systems design. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2019.*

Jose J. Gonzalez, Leire Labaka, Starr Roxanne Hiltz, and Murray Turoff. 2016. Insights from a Simulation Model of Disaster Response: Generalization and Action Points. In *HICSS '16: Proceedings of the 2016 49th Hawaii International Conference on System Sciences.* IEEE, Piscataway, NJ, 152–161. https://doi.org/10.1109/HICSS.2016.27

Juvare-Inc. 2020. *WebEOC.* http://juvare.com/webeoc

Jun Kato. 2017. User Interfaces for Live Programming. Keynote presentation at LIVE 2017, Vancouver, Canada.

Jun Kato and Masataka Goto. 2016. Live Tuning: Expanding live programming benefits to non-programmers. In *Proceedings of ECOOP LIVE.* ACM. https://junkato.jp/live-tuning/

Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software.* ACM, 53–62.

Karen Neville, Sheila O'Riordan, Andrew Pope, Marion Rauner, Maria Rochford, Martina Madden, James Sweeney, Alexander Nussbaumer, Nora McCarthy, and Cian O'Brien. 2016. Towards the development of a decision support system for multi-agency decision-making during cross-border emergencies. *Journal of Decision Systems* 25, sup1 (2016), 381–396. https://doi.org/10.1080/12460125.2016.1187393 arXiv:https://doi.org/10.1080/12460125.2016.1187393

Jongsoon Park, Ralph Cullen, and Tonya Smith-Jackson. 2014. Designing a Decision Support System for Disaster Management and Recovery. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. SAGE Journals, 1993–1997.

Tomas Petricek. 2019. *Histogram: You have to know the past to understand the present.* University of Kent. http://tomasp.net/histogram/

Hans J. Scholl. 2019. Overwhelmed by brute force of nature: First response management in the wake of a catastrophic incident. In *EGOV 2019*, I. Lindgren et al (Ed.). Vol. LNCS 11685. Springer Nature Switzerland AG, 105–124.

Steven Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*. IEEE Computer Society, Los Alamitos, CA, 31–34.

Dave Thomas. 1997. *Travels with SmallTalk.* http://www.mojowire.com/TravelsWithSmalltalk/DaveThomas-TravelsWithSmalltalk.htm

Murray Turoff. 2002. Past and future emergency response information systems. *Communications of the A.C.M.* 45 (2002), 19–32. Issue 4.

Bret Victor. 2012. Inventing on Principle. video. https://vimeo.com/36579366

World-Health-Organization. 2013. A Systematic Review of Public Health Emergency Operation Centers (EOC). http://apps.who.int/iris/bitstream/10665/99043/1/WHO_HSE_GCR_2014.1_eng.pdf

Xerox-PARC. 1987. The Analyst Workstation System. In *Xerox Special Information Systems.*