Reading Between the Lines of Code: Visualising a Program's Lifetime

Nikolai Sultana

Department of Computer Science and AI,
University of Malta

Abstract. Visual representations of systems or processes are rife in all fields of science and engineering due to the concise yet effusive descriptions such representations convey. Humans' pervasive tendency to visualise has led to various methods being evolved through the years to represent different aspects of software. However visualising running software has been fraught with the challenges of providing a meaningful representation of a process which is stripped of meaningful cues and reduced to manipulating values and the field has consequently evolved very slowly. Visualising running software is particularly useful for analysing the behaviour of software (e.g. software written to make use of late binding) and to gain a better understanding of the ever-important assessment of how well the final product is fulfilling the initial request. This paper discusses the significance of gaining improved insight into a program's lifetime and demonstrates how attributing a geometric sense to the design of computer languages can serve to make it easier to visualise the execution of software by shifting the focus of semantics towards the spatial organisation of program parts.

1 Introduction

Humans find it easy and effective to communicate using drawings — both between themselves and increasingly with computers. Visualisation is heavily used when designing software however we lack the means of visualising a running program in a way that enables us to see the software working in a tangible manner. Such a feature of software would need to be added at an extra effort since code is typically written to work well and not to look good while working. A visual aspect to the program's workings would need to be explicitly created and would not be a natural part of the code hence potentially leading to a gap between the code and the visual representation of its execution.

In certain programmings languages circuit diagrams are used to convey a structure of how components connect and work together to form the program — this in itself lends itself to a notion of visualising what's going on in a program while its running (if the information about program components and interconnections is retained when the code is translated and executed, which is usually not the case).

The motivations of this research revolve around the desire to have richer tools for software developers, tools which are compatible with human cognitive processes and traits in order to improve qualities such as usability and (especially) sustainability in software development. It is widely held that the largest catalyst for the use of visual representations is that humans are quick to grasp the significance of images and reason about them. We tend to prefer to conceive parsimonious visual descriptions conveying elements of information which serve to enrich a representation by abating ambiguity. A lot of motivation for this paper was drawn from [1] in which, apart from the ubiquitously known argument against use of the go to statement, a remark was made about the correspondence between the running program and the code which controls it. This was followed

by another remark about the fairly primitive ways of visualising processes evolving in time available. After almost 40 years our methods and tools have still remained rather primitive, especially compared to the advancements made in the stages leading to the implementation of software. The difference of computing power available in 1968 and now certainly plays a role since to enrich one's experience of computing one requires more computing power.

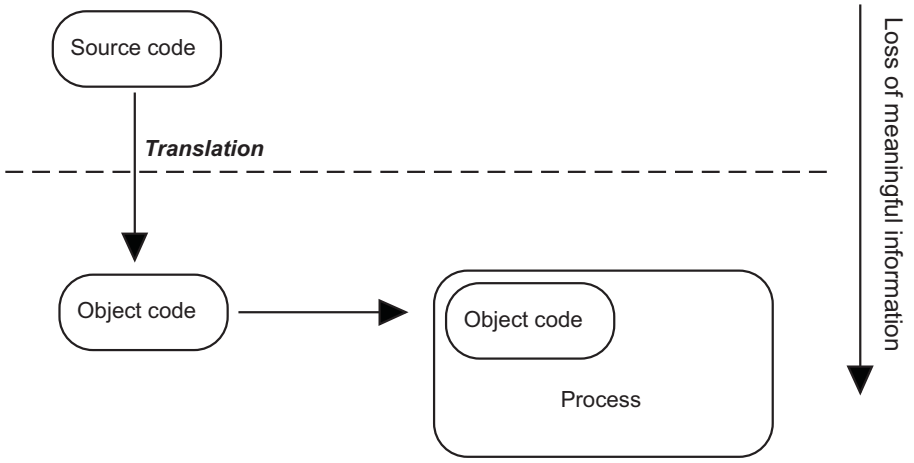
This paper attempts to address the concern of impoverished means of visualising running programs and Dijkstra's appeal to diminish the conceptual gap between the program and the process. By a program's lifetime we understand the time interval spanning a program's execution and not the path from inception leading to implementation and maintenance of a program since we believe that this gives the actual execution of a program the importance it deserves above other stages leading to or following its deployment.

It is generally (and overly generously) assumed that the implementation of software will follow its design. This gap between visualising the design and experiencing the implementation can be partly alleviated by having a means of reasoning about the system's execution in a visual fashion, allowing us (or another system for that matter) to follow the execution and observe discrepancies between our expectations and the program's behaviour. The importance of paying special attention to the quality of the implementation is not only due to it being an artefact which is the product in the software value-chain with which the client has most contact, but [2] presented results indicating that even though rigorous formal techniques are applied during the design of software it is not enough to eradicate problems in the implementation. The implementation of a system deserves more importance than being seen as one of a number of stages constituting the lifecycle of a piece of software, but rather be seen as standing out as the process which delicately hinges on to extending and abiding to other pieces of software which themselves might contain inconsistencies and faults and attempting to insulate as much as possible the end-user from shortcomings in software which talks to our software.

The complexity of software is increased primarily because of the increasing demands being made on it. This complexity tends to cloud designers' foresight — some results can only be predicted with a lot of difficulty due to the large number of interactions carried out by various systems. The dense amount of interactions renders it hard to ascertain what sequence of states could render a system or parts of it non-functional or have them return erroneous results.

Another motivation in this research is that of having an account of the lifetime of a piece of software (analogous to a detailed autobiography) which can be analysed by humans or other software in the event of any failure or to review a program's behaviour (e.g. to improve its performance).

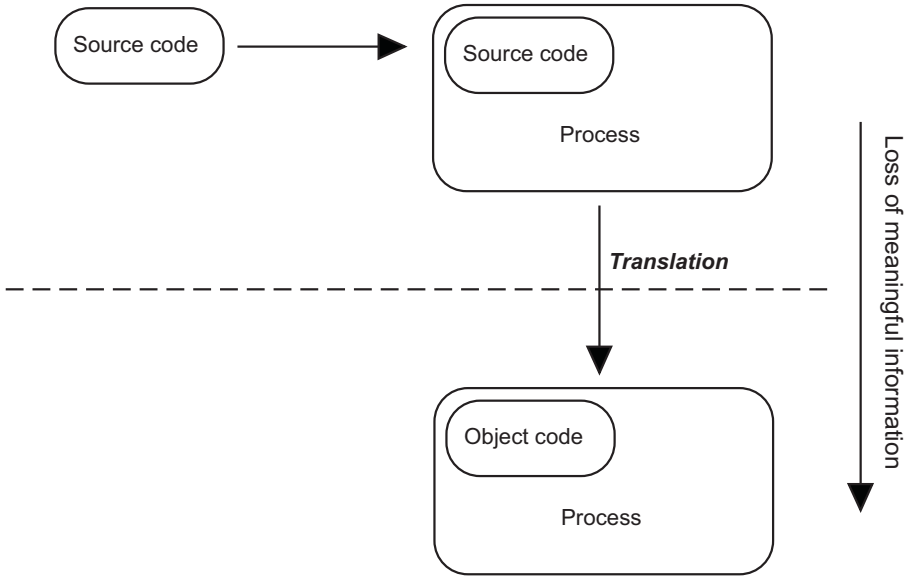
As in any communication channel, a software development process suffers form the arbitrary introduction of noise from shortcomings in the communication between the various stakeholders, making it harder for the implementation to be faithful to the design. Moreover the nature of executing software makes it more difficult to visualise due to it being stripped of symbolic cues about its overall function during translation to a lower level. When software is translated features of the text which render the code readable are removed from the object code. This is understandable since for the execution of the software such information is not required. Information lost in this manner is not retrievable if the translation process were to be reversed (and often such a reversal would return a slightly different of the code if any optimisation were applied by the compiler). Understanding the program's lifetime is more difficult due to the impoverished information we can be provided with and this challenges our ability to excogitate the details of the program's operation.



Certain ways of dealing with the translation of code change the organisation of the diagram shown above by leaving traces of information in intermediate positions along the path which leads to the production of object code. The primary example of this are some of the systems which generate intermediate code which is then executed by a virtual machine — this information can be used to reason about the program's execution in a more meaningful manner (e.g. represent objects and their relationships, methods invoked on them, etc). The intuitive wealth of such descriptions of the program's lifetime is still limited nonetheless and a number of ideas will be presented later on which attempt to push more of the program's execution towards the information-rich side of the representation rubicon.

The Bubble programming language will be used to demonstrate an approach towards computation by including spatial organisation of mobile program components and their relationships in the decisions taken to arrive to the program's behaviour. A different model for running software is used in Bubble's execution environment, which retains information about the program's lifetime and makes it very easy to visualise and reason about.

This paper proceeds to review the widespread notion of readability in programming, outline the state of affairs in runtime inspection of software and its interplay with visualising programs in execution in Section 2. In Section 3 a number of features are identified which would equip a programming language with basic properties to be used to express programs of which execution would be intertwined with their visualisation. Section 4 outlines the implementation of such a language and refers to a case study, moving it to the context of software visualisation since that is the aspect of interest in this paper. Section 5 briefly describes a tool written to generate a graphical representation of the description of a program's lifetime and Section 6 presents a short contrast of advantages and drawbacks of the ideas and approach put forward in this research, together with identifying areas which could benefit or benefit already from work in software visualisation.



2 Background

2.1 Program readability

A fair amount is emphasis is placed on the importance of producing readable code when teaching programming skills since this improves its maintainability, affects somewhat its re-usability as well as the total cost of ownership of the software — instilling such a sense at the very beginning of exposure to programming makes plenty of sense. Readability serves to assist the perception of the person reading the code (which could be the original author of the code) in interpreting what it does. It is commonly associated with indenting code pertaining to particular blocks, adding in-line comments describing in a natural language what certain bits of the code do and using meaningful strings of characters as identifiers. Readability is also influenced by the choice of operators used and the amount of space used for the expression of the solution to a problem (with obfuscated code usually being on the minimalist side of expression).

Due to the way software is translated it is rarely the case that much thought is given to a notion of process readability since a process (program in execution) is conventionally not for humans to control, but rather handled entirely by the machine during a program's lifetime as it manages its internal workings. Process readability is not addressed by traditional methods of eliciting information about running programs (runtime inspection) because of there not being a ubiquitous system of assigning execution of code with an appearance. In spite of that the appearance and its interpretation is of utmost importance when it comes to visualisation and having the execution of code being a function of an appearance it operates on would allow humans to create more analogies between pictures and operations.

The importance of producing readable code is supported by awareness about our capabilities of cognition — for example [3] had hypothesised that we can keep track of around seven things at the same time. This made it into programming as the Hrair limit and serves to curb the amount of subroutines called from the main program and fundamentally results in facilitating visualising the program when reading the code (since there are less things to track).

2.2 Eliciting information about program execution

A visualisation pipeline usually involves gathering data as the first step. Data is then processed and filtered, rendered and displayed. Conventionally the quality of information we can elicit about a running program is quite poor — it does show what the program is doing and how it is doing it but does so in an abstruse fashion.

We can gain insight about a running program (dynamic analysis) in a variety of ways. Initially one can look at a program's memory and processor-time usage. We can gain a lot more information by looking at its communication with the rest of the system — through files, sockets, and other channels (even UI). However, such information tends to be overwhelming and unwieldy since its contents can be arbitrary and were not intended for a visualisation of the running program to be produced — the intention behind them was just to get the program working. A variety of tools can be used to monitor the internal workings of a program, but very often the information accessed in this manner is too fine grained and not intuitive — thus it doesn't convey the big picture about what the program (or significantly large chunks of it) is doing. An often used technique is inserting diagnostic code (e.g. in the form of statements sending values to standard output) to obtain an indication of the values of certain variables during the program's lifetime. Software can be set to send information to a system-wide logging facility (e.g. SysLog) or have their own logging mechanism (e.g. this can be created with ease in AOP as an aspect weaved to an application).

As previously mentioned, a generous interpretation of visualisation of program run-time allows us to consider what we see on screen and what is dumped to file/communication channels as pertinent information about what the program is doing. This contains what we choose to show (e.g. to adhere to specification of software, protocols, etc) and typically offers poor quality of information in terms of what's going on inside due to the way software is/was written to carry things out rather than carry things out by showing it is doing so.

More pervasive means of inspecting and visualising program lifetime are required. This stems from the way software is written and/or expected to be written. An inherent geometric sense would lend itself well to human visualisation abilities by instilling a sense of spatial coherence in the writing of software that will manifest itself in its workings and will take a role in its execution. The following section elaborates on this point.

2.3 Software visualisation

Giving software an inherent appearance laden with information about its operation would consolidate it with its modelling aspect and empower stakeholders of software to take better decisions based on the operation of software. Traditionally a variety of graphs have been used to describe the lifetime of a program. Call graphs, control flow graphs and action diagrams are all depictions of the parts of a program which were called while it was running and helps us understand better the sequence of events which characterised the program's execution as well as offering the possibility of aggregating information into a profile. A profile is used to describe an aspect of a system (e.g. time spent executing in a part of a program or else representation of the distribution of statements in a program). Profiling therefore provides us with a summary of the program's execution but the information returned is often too general to deal with specific details.

Learning what's being done in running programs largely depends on the paradigm those programs were written in: it tends to be the case that the more sophisticated the paradigm (in terms of metaphors with the real world) the more meaningful the representations. For example, an execution trace can usually be produced for programs written in any language, but diagrams involving relationships between objects can be produced for programs written in OO languages.

A variety of tools exist which produce representations of program's lifetime for programs written in a variety of languages (e.g. [4–6]). Other tools depart from the source code and provide the user with support for software comprehension [7, 8]. Some tools are designed to augment existing tools and adding new features [9]. The link between software visualisation and cognition has been discussed in [10] together with an approach that attempts to support the latter to arrive at the former.

The way that software is written does make a difference to the representation — for example different approaches to a program's modularity will return different call graphs which would usually be of different complexity (in terms of clutter).

3 Programming Spaces

The primary concern in programming deals with getting a working program (which fulfils the requirements which caused it to be written). No thought is given to the program having any inherent visual significance and as previously mentioned information describing the program (albeit limited) is lost during translation and this further limits our ability to visualise running programs in a meaningful manner. Program components are usually designed in such a way that they have to be (at least) conceptually linked to be useful. The components themselves have no location and therefore the information conveyed lacks the relationship between the function and location of elements. If a more geometric view of programming is taken the program components can move and interact and would function according to their location. This pitches computation at a level where spatial organisation and behaviour are related, hence one can influence the other. This would encourage a mechanistic view of computation and allow transferring somewhat visual representations from a description of the problem to a program's implementation.

It follows that a program can therefore be thought of as a description of points (instances of certain types) moving in a space and the execution of the program involves the properties and proximity of such points — thus programming would consist of conveying the description of a system into a computer representation in a way that retains positional information. This information would also contribute to visualising the execution of the program in a meaningful manner.

The semantics of a programming language would need to cater for this information, but the benefit of this would be that the semantics would not only have a functional but also convey a visual concern.

4 The shape of a program

Bubble [11] is a language which was developed to model certain natural phenomena [12], transferring descriptions of aspects of our perceptions of such phenomena into a form which could then be used in a simulation. A framework handles the classification of roles and the ways of reproducing (an approximation of) certain dynamics as observed in nature. This language was also used to look at some ideas in Computer Science and a report [13] was prepared about different implementations of a Turing Machine using Bubble. This case will be used as an example of a simple example of a visualisation produced for a program lifetime and a statement about the suitability of this approach for different genres of programs will be made further down.

In Bubble a program is composed of entities having properties amongst which is a position in a space. The program's initiation is a process which involves producing and laying out such structures in the space and then a message can be passed to the execution environment instructing it

to start the program. The program's execution revolves around determining the truth value of certain triggers and then carrying out the appropriate transformation on the space. Triggers involve determining the satisfaction of conditions including property values and/or the composition of the neighbourhood of an object.

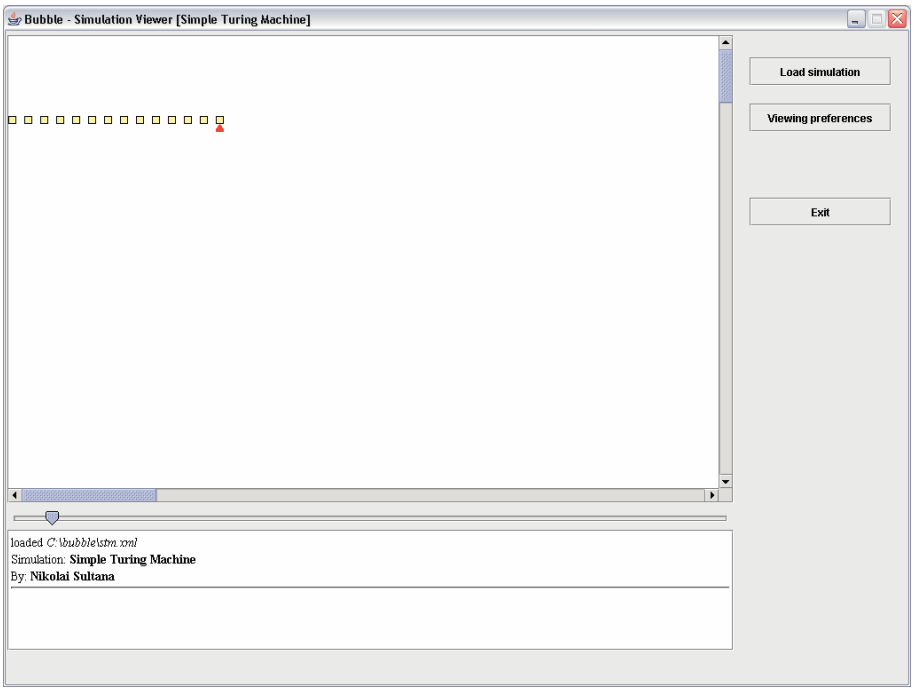
The end result of implementing a program in Bubble is that a program would consist of a distribution of points each with a potential for carrying out an execution. As the program executed the execution environment keeps track of the changes made to the space in which the program is situated and a description of its lifetime can be produced in XML. This can then be transformed using appropriate tools or else passed on to a specifically written tool for visualisation.

The preferred implementation of the Turing Machine in [13] treats two kinds of objects (bits of tape and the read-write head) and the initialisation sets up the tape and read-write head to be at the start position. The behaviour of the head follows a state-transition diagram by specifying the sensitivity of the head to the presence of the tape and its contents and these conditions trigger changes carried out inside the head (current state) or space (movement).

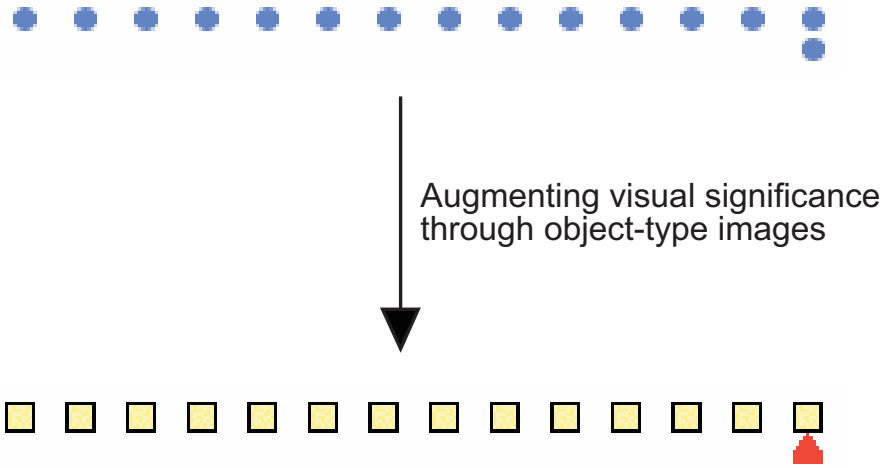
While the specific details of the implementation are provided in [13] the final result of writing the program in Bubble was having a program which worked by manipulating its spatial organisation. Information could be elicited about each stage of the program and the next section will describe a tool used to render this information into a more visually appealing form.

5 Visualising the Program's Autobiography

Once a description of a program's lifetime has been generated it is fed to a viewing tool written specifically to interpret it and present the user with a visual representation. A time-line is provided for the user to manipulate and move forwards and backwards in the lifetime of the program. The program's components are represented on the screen and their behaviour can be observed as the user moves across the time-line. Different types of components can be represented using different user-specified images, making it easier to understand what is occurring since the movement of the components is augmented with a pictorial representation pertinent to the nature of the component. Clicking on any component returns details about that component (its properties and their values).



This visualisation of the program's lifetime is only one of several possible processes which can be applied to that information — future work can seek ways of refining the information, filtering it, or possibly create an animation of the interactions between objects since the XML description contains certain information which the current visualisation software does not utilise to the fullest. The aim of this software is to show the possibilities of this approach to programming and attempt to shed some light on its suitability. Zooming into the system to observe its components of components will be required for complex systems and visual cues can be used to convey information to the user and enrich their perspective on the software's operation.



Bubble allows annotations to be added into the code and these will be automatically reproduced in appropriate locations in the description of the program's lifetime. These annotations are displayed according to the current point in the time-line the user is viewing in order to provide more information (as included by the programmer) about the context of that point in the program's lifetime.

6 Gains and Drawbacks

The advantage of having a representation of a program's lifetime adds value to the stakeholders of software since it empowers them with insight about the software. Writing software can be easier since it is more clear what is going on when the software is being executed — people can *see* it working. Another advantage of having a rich description of a program's behaviour is that tools can be developed to evaluate and assess the software's behaviour and performance in an automated fashion. In the case of a software failure the representation can be used to examine what led to the surfacing of the failure and take better decisions about fixing the fault.

Since a different (and not commonly used) programming language and execution environment where used in this research there is the disadvantage of the artefacts not being directly applicable to any program. It's also true that producing a visualisation involves a cost: executing and visualising software in this manner, although beneficial, adds a burden on its performance and slows it down. Furthermore, this approach to programming might not be ideal for certain genres of programming, though on the other hand it's always best to use languages inclined to a particular application to get the job done better (or having the qualities to render this more likely). Not having a silver bullet for software [14] also includes not having a single programming language which is apt for any problem: having a variety of languages is beneficial and allows formulating better solutions to more problems. Another criticism of this research might be that a Turing Machine is in itself highly mechanical and might not have served as the best example to use for software visualisation. However many algorithms tend to operate in a mechanical fashion, even if this is not immediately evident, or could be adapted to work that way (rendering them more intuitively comprehensible in the process).

Progress in software visualisation is inclined to produce benefits in diverse areas by allowing one to look at certain problems from new angles. Examples of applications of the information once a reasonable automated description of a program's lifetime has been generated are the following:

- Testing and debugging, preventive and corrective maintenance of software
- Simulations
- Understanding interaction between/within software
- Reflection Reverse engineering Teaching programming and computer literacy
- Refactoring

7 Conclusion

The duration of our engagement with the a formulation of a program usually lasts until we compile and run the software. Then testing is carried out to see how well our expectations are met. A dire need exists for more tools and techniques to extend our contact with the domain of processes in order to better understand what goes on during a program's lifetime, why specific failures occur, etc. While visual representations of a program's functioning are used frequently in the stages leading to its implementation very little visualisation is carried out on its execution.

In this research a lot of effort was made to avoid reinventing the wheel and instead approach the problem from identified shortcomings in the nature of software and moving towards software operating as a function of its appearance. Due to the way things evolved in the history of programming there wasn't sufficient opportunity of thinking things through (certain concepts *grew* that way) and it might be of benefit certain ideas are re-evaluated.

Dealing with with the heterogeneity in programming languages makes it difficult to create general purpose visualisation tools for program execution. The goals outlined in the beginning of the paper were met by the approach to programming outlined in the paper, but the cost involved is very high due to lack of alternatives for creating such programs. This makes it difficult to apply this approach to mainstream languages. However, this might be alleviated by focussing on the intermediate level and having the translation process involve creating geometric representations of programs in languages before going on to generate low-information representations of the code (i.e. first talking to a form of middleware, then the code will be translated to a lower level representation).

We believe that elegance in software design can be improved by converging execution of code with a meaningful spatial representation of the running program. This adds another dimension to programming since along with readable code and an intuitive interface the programmer must ensure that the program executes in such a way that the motion and interaction of its components make sense. The readability of code is somewhat extended by this concern since it also covers the dynamic aspect of executing code. Having a description of a program's lifetime and being able to translate that into pictures has several advantages and serve to increase the tangibility of software, which in turn offers plenty of appeal to human cognition. Having a representation of a running program which is compatible with the way we work should offer a good opportunity to understand such a program by giving us the means of looking at it through new eyes.

References

1. Dijkstra, E. Go To Statement Considered Harmful. In: Communications of the ACM, Vol. 11 (March 1968) 147-148.
2. Pfleeger, S., Hatton, L.: Do formal methods really work? IEEE Computer (January 1997).
3. Miller, G.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. The Psychological Review (1956), Vol. 63, 81-97.
4. Hopfner, M., Seipel, D., Wolff von Gudenberg, J.: Comprehending and Visualizing Software Based on XML-Representations and Call Graphs. Proceedings of the 11th IEEE International Workshop on Program Comprehension (2003).
5. Bertuli, R., Ducasse, S., Lanza, M.: Run-Time Information Visualization for Understanding Object-Oriented Systems. 4th International Workshop on Object-Oriented Reengineering (2003) 10-19.
6. Systa, T.: Understanding the Behavior of Java Programs. Proceedings of the Seventh Working Conference on Reverse Engineering. IEEE Computer Society (2000).
7. Wu, J., Storey, M-A.: A Multi-Perspective Software Visualization Environment. Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research.
8. Storey, M-A., Muller, H., Wong, K.: Manipulating and Documenting Software Structures. Proceedings of the 1995 International Conference on Software Maintenance.
9. Zeller, A., Lutkehaus, D.: DDD-A Free Graphical Front-End for UNIX Debuggers.
10. Eduard Todureanu, M.: Designing effective program visualization tools for reducing user's cognitive effort. Proceedings of the 2003 ACM symposium on Software visualization.
11. Sultana, N.: Bubble Language Specification (2004).
12. Sultana, N.: Modelling and Simulating Systems using Molecule Interactions, Technical report, University of Kent (2004).
13. Sultana, N.: Bubble Case Study — a simple Turing Machine (2004).
14. Brooks, F.: No Silver Bullet: Essence and Accidents of Software Engineering. Computer Magazine, April 1987.