Programming Experience Requirements for Future Visual Development Environments

Anthony Savidis[1,2]
[1]Institute of Computer Science, FORTH, Heraklion, Crete, Greece
[2]Department of Computer Science, University of Crete, Greece

Keywords: Visual Programming, Learning Programming, User / Learning Experience, Development Environments.

Abstract: Visual programming is widely adopted for teaching purposes, considered as an appropriate starting base before introducing learners to typical programming languages. However, the progress in such tools is very slow and limited compared to standard programming environments. Moreover, there is no systematic classification regarding the most important requirements to improve the support of visual programming tasks. In this context, we introduce programming experience as the context-specific notion of user-experience for the programming domain. Then, we identify three groups of requirements relating to language, interaction and tools, and elaborate with specific requirements per group. In this analysis, we study related examples from current tools in various domains, while we propose scenarios inspired from source-based programming environments.

1 INTRODUCTION

The notion of visual programming concerns methods to define programs in a multi-dimensional fashion (Myers, 1990). The latter is not linked to the underlying program representation, but concerns the interactive visual means through which a program is created, refined and managed. Hence, text-based code is considered as one-dimension method and is therefore not treated as visual programming.

While visual programming adoption ranges from rapid application development, interactive software configurations, and system administration, it became popular for educational purposes, in particular for teaching programming skills. In this framework, Scratch (Maloney et al., 2010), a block-based tool and Lego Mindstorms™ (Vallance et al., 2009) are amongst the most well-known visual tools in learning contexts. Historically, visual programing systems have been deployed to introduce students in the programming universe before being enabled to manage and master professional source-based programming languages. In this context, their scope is generally considered to be restricted in the early stages of acquiring programming skills. But today, there are visual tools for professional development purposes, ranging from business process, Internet of Things, 3d graphics and robotics, meaning their scope is not merely restricted to learning activities.

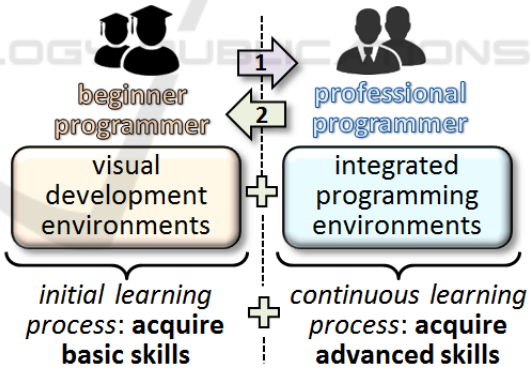

Figure 1: Adoption of visual development tools for continuous learning in professional programming.

Also, such tools support an important activity in the development lifecycle that is not always substituted by text-based counterparts, thus retaining a distinct and critical role. In this sense, for certain development skills, professional programmers may still have to learn using visual tools before switching (if they ever do) to the most powerful programming-language basis. This interplay between visual tools for leaning, with typical professional programming environments, is depicted under Figure 1.

Savidis, A.
Programming Experience Requirements for Future Visual Development Environments.
DOI: 10.5220/0011082500003182
In *Proceedings of the 14th International Conference on Computer Supported Education (CSEDU 2022) - Volume 2*, pages 284-292
ISBN: 978-989-758-562-3; ISSN: 2184-5026

In an educational context, emphasis is put on blending user experience (Law et al., 2009) with learning experience (Tawfik et al., 2021) to optimally support programming tasks. We define this combination as *programming experience* (see Figure 2) to better highlight and contextualise the importance of the programming task.

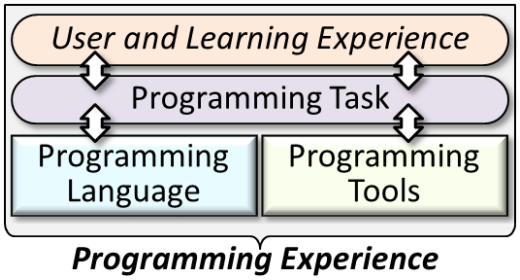

Figure 2: Programming experience as the overall user and learning experience in programming-related activities.

In this context, we carried out a systematic analysis briefed in this paper, resulting in key design requirements linking to programming experience, with a summary provided under Figure 3.

1. *Language* Requirements
• Explicit Language Paradigm
• Visible Syntax and Semantics
• Intelligent Editing Automations
• Extra Optional Elements
2. *Interaction* Requirements
• Appropriate Element Metaphor
• Reasonable Visual Complexity
• Configurable Level of Detail
• Extensible Code Annotations
3. *Tool* Requirements
• Project Management
• Debugging Facilities
• Programming Assistants
• Custom Static Analyzers

Figure 3: Overview of the list of programming experience requirements for visual development environments.

2 RELATED WORK

In (Kiper et al., 1997) there is one of the earliest taxonomies with criteria judging visual languages, in particular: visual nature, functionality, ease of comprehension, paradigm support and scalability. Although the analysis is outdated, it is historically the first systematic effort in setting specific driving principles for visual programming systems.

In (McGuffin & Fuhrman, 2020), the focus is shifted on the classification of visual programming techniques rather than on the design requirements.

In (Repenning, 2017), although no design requirements are negotiated, it is important to note the critique regarding the lack of semantic tools like context-sensitive pragmatic explanations (mostly evaluation time) that would improve the user experience.

3 LANGUAGE REQUIREMENTS

This category concerns the linkage to an underlying programming language category, fully or partially. It is the theoretical foundation of the visual programming system, the formal backend for which the visual system provides a friendlier fronted.

3.1 Explicit Language Paradigm

The visual programming system should rely on an explicit underlying theoretically-oriented language paradigm, sometimes carefully chosen combinations of paradigms such as: imperative, object-oriented, event-driven, flow-based, batch processing, functional programming, message passing, constrained systems, etc.

Clearly, it is crucial to document and explain the primary reason a specific paradigm is chosen for the target learner audience and domain (if one or some are explicit targeted), setting an anticipated learning curve and engineering criteria like: ease-of-use, intuitive deployment, rapid development, error prevention, engineering scalability, proximity to a real language that learners might have to use, etc.

Once the paradigm is selected, designers may optionally expose ideas and techniques regarding the way the primary programming elements of the paradigm may be mapped to visual counterparts, and the reason such a mapping is considered appropriate and consistent. In certain cases, when visual designs aim for specific purposes or tasks, even if the underlying language paradigm is of general purpose, it has to be explicitly stated and justified.

Examples: The most common programming language paradigm is the *imperative*, with variables, assignments, statements and expressions, and the *flow-based*, blending functional characteristics with event-oriented elements. In particular, Scratch offers blocks reflecting a purely imperative paradigm, while Lego Mindstorms Ev3 (Vallance et al., 2009)

provides elements with an imperative look, but a genuine functional style. Business Process Modelling and Notation (BPMN, Ko et al., 2009) is a domain-specific visual language that adopts the flow-based programming paradigm (FBP, Morrisson, 1994). The roots of FBP are in control-flow and stream processing, while it borrows elements from functional composition, batch processing and event-driven systems.

In general, we refer to such cases as *visual syntactic illusions* when the graphical language conveys an underlying paradigm but with a frontend that mimics alternative more familiar paradigms.

Touch Develop by Microsoft (Ball et al., 2016) combined the imperative programming style with typical object-based elements (not including any class definition or inheritance features), while offering the syntactic illusion of message-passing regarding method invocations.

Finally, Blockly by Google (Pasternak et al., 2017) offers an API to create new types of blocks and to map them semantically to an underlying implementation. For example, it is common to introduce custom blocks combining the object-based and event-based paradigms.

3.2 Visible Syntax and Semantics

Visual languages adopted for learning purposes become stepping stones to acquire more advanced skills and further exploit the expressive power of a real language. Essentially, visual languages are blankets which hide or abstract the underlying complexity of the real language and its detailed programming model. However, this does not imply that the syntax and semantics of the backend language should not be transferred in the visual frontend. Completely separating those two, not only makes the transition to the real language harder, but may reduce the chances to educationally deliver fundamental concepts that are only present within the original language syntax and semantics. In particular, *visual syntax concerns all geometric and graphical rules applying to visual program composition that also map directly to the grammatical elements of the underlying language*.

As mentioned earlier, due to syntactic illusions, this mapping need not be very precise. However, it is crucial that the visual language is delivered in a way that its visual syntax and the corresponding semantics are explicitly and naturally mapped, linking unambiguously to backend language semantics. Elements like recursion, repetition, nesting and scoping should be appropriately represented by corresponding visual structures.

Examples: Spreadsheets are treated as visual programming systems with their backend relying on grid formulas and constraints. Theoretically, formulas reflect a functional style that users directly deploy over a visual grid, with interactive facilities to refer to grid cells, individually or collectively, via pick or group selection. The syntax and semantics of the underlying language are completely visible, implying *spreadsheets are less visual languages and more scripting systems on grid elements*.

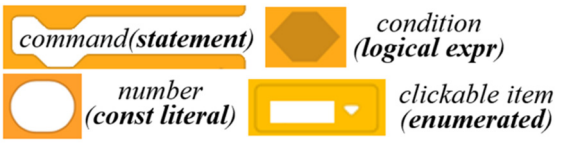

Figure 4: Scratch visual syntax relies on geometric shapes, however, it conveys no other syntactic information.

Scratch (Maloney et al., 2010) adopts a concise visual syntax, with encoding of elements and placeholders (as outlined under Figure 4) that maps directly to the text language. However, there is a simplification and potential information loss, since sub-categories of expressions and statements all map to just a single syntactic visual symbol.

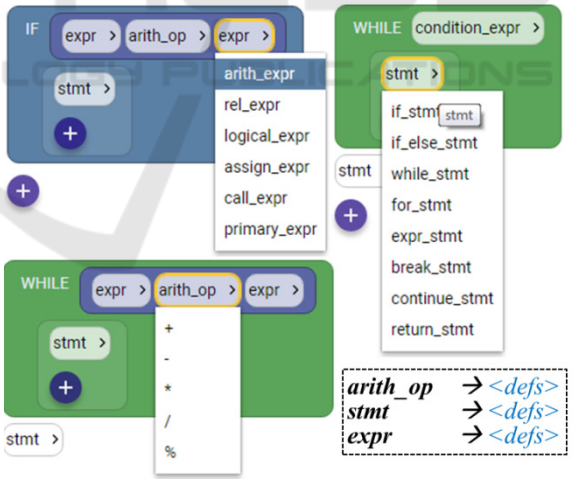

Figure 5: Code-Chips syntax-directed definition, besides drag-n-drop blocks, makes the underlying grammar explicit in block-based program composition.

An increasingly popular visual system is Node-RED (Open JS Foundation, 2022), adopting the flow-based paradigm that hides the underlying details of event-driven programming via consumer-producer chains. As a result, the syntax of event blocks is hidden, while event management is

semantically treated as data flows. But still to introduce new custom nodes, one should directly program them in JavaScript. Overall, learners using exclusively Node-RED may never gain a deep understanding of asynchronous event management. Clearly, this is a well-known tradeoff between expressive power and ease-of-use. Finally, a tool unifying the visual and text syntax is presented in (Agapakis, 2021), supporting syntax-directed editing combined with interactive blocks (see Figure 5). It offers interactive syntax, making learners aware of the grammatical profile of every program element.

3.3 Intelligent Editing Automations

Editing automations are commonly referred to as *IntelliSense*, including at least four key features: quick info, go-to definition, auto completion and parameter help in function calls. The benefits of such automations go beyond rapid development, enabling better understanding of the source code.

While typical dropdown suggestions for fields and methods are integrated in most visual tools like Scratch, Blockly and Touch Develop, quick info, that is available in professional development tools, is not fully included. For visual programming, auto completion features may be extended to support: (i) context-sensitive element suggestion; and (ii) guided or assisted visual element composition.

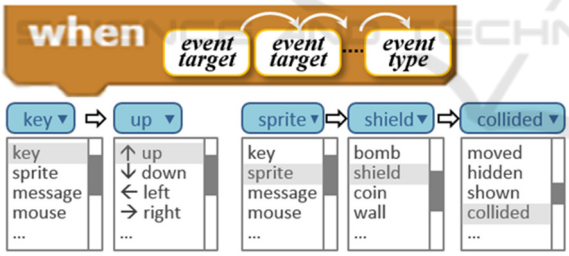

Figure 6: Mockup for auto-completion in *when* blocks, with automatic suggestion of variable event-target placeholders with successive dropdown lists.

Examples: The mock-up of Figure 6 resolves the necessity of multiple distinct "*when*" blocks in Scratch, per event type and target, with just a single block. It depicts event target specifiers, defining the event target expression, followed at the end by the actual event type.

For example, in 2d games, sprite targets require firstly the main category "*sprite*" and then the sprite class, like "*shield*", with the contextually applicable event types suggested via a dropdown list.

Another mockup, illustrated under Figure 7, concerns flow-based programming, and suggests the provision of adaptive tooltips between connected elements, carrying brief information on the type of propagated data items. When interactive updates change the output data types, all outdated visual cues must be automatically refreshed and synced. Quick information is usually provided as informative tooltips, something known to encourage users in interactively exploring features and functionality.

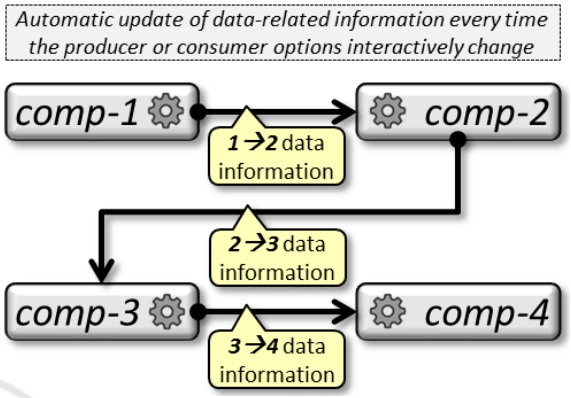

Figure 7: Mockup for auto-completion of data-related information (auto callouts) in consumer-producer flows.

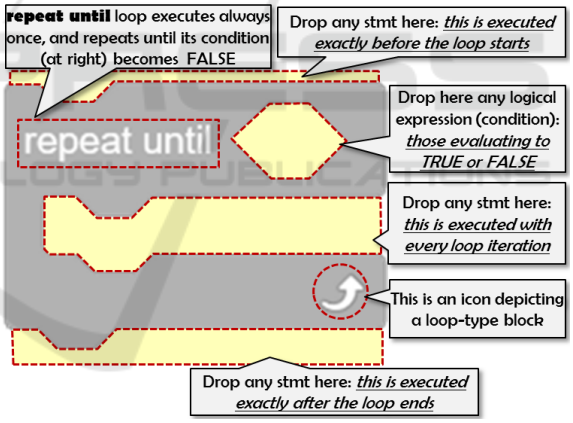

Figure 8: Hot areas for tooltips (shown as callouts) in Scratch blocks, providing semantic information to better support learning programming during visual-code editing.

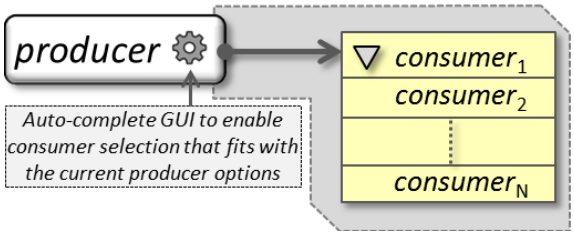

Figure 9: Mockup for auto-extension by suggesting consumer types matching a selected producer block.

In Figure 8 we show how block-level editors may introduce tooltips to inform learners in using programming elements. *Such tooltips offer a live, brief, and interactive tutorial of the visual language*. In Figure 9, in flow-based programming, if a producer is selected, an appropriate dropdown list of matching consumers is composed and shown, based on the type of the output port and the information on the input ports of all available consumer types.

3.4 Extra Optional Elements

Being learning tools, visual programming systems strike for a balance between programming facilities and ease-of-use. Thus, more advanced programming techniques may be left out, restricting deployment to small-scale projects. In this context, optional programming features may be provided targeted to more advanced learners as listed below:

- *Scopes* may be defined by visual grouping, with local variables being geometrically contained.
- *Modules* allow split programs into multiple units and reuse them in one or more projects.
- *Hybrid code* enables contrast the visual code and its respective textual form altogether, enabling mixed editing, while keeping both views fully-synced and well-formed.
- *Source code* that complements visual code, helpful for implementing complex modules, and also for supporting the cooperation of learners with experienced programmers.

4 INTERACTION EQUIREMENTS

4.1 Appropriate Element Metaphor

In visual programming there is always a primary geometric element with some degree of graphical variability and styling. For instance, it is a *jigsaw puzzle* block in Scratch and Blockly, a game *card* in Kodu (MacLaurin, 2011), a *consumer-producer* block in Node-RED, a *process* block in BPMN, a *circuit* block in LEGO Mindstorms Eve3, and a command *block* in Touch Develop. This primary element plays a crucial role, since its choice and representation, commonly as a real-world world metaphor, is extremely important for the quality of the programming experience. To assume a single visual element is capable to model all program elements is questionable and likely optimistic.

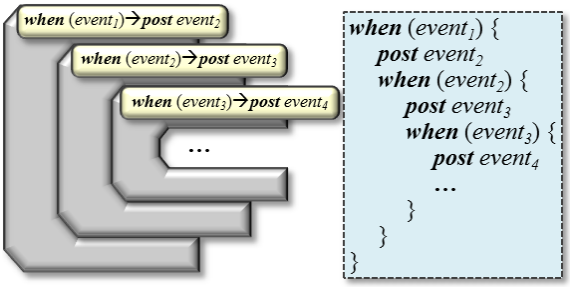

Figure 10: Mismatch of blocks (left) implying scope nesting (right) and the event-driven paradigm.

In fact, there is no such analogy in programming languages, since elements with distinct and diverse semantics like classes, functions, expressions and variables coexist, but with notable structural differences at the source-code definition level.

Also, a potential mismatch may occur if the visual structures on the graphical domain differ significantly from the implied semantic structures in the underlying programing language paradigm.

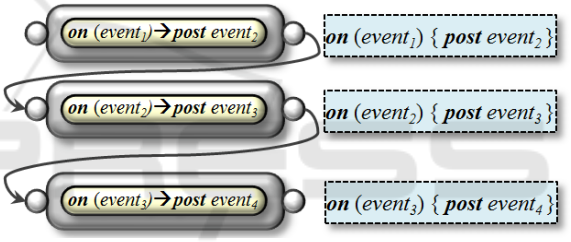

Figure 11: The previous example in a flow-based metaphor better matches the model of cascaded scope-free event-based consumer-producer handlers.

For instance, consider consumer-producer chains in event management using jigsaw blocks. With independent *when* blocks (as in Scratch) dependencies are not shown as linked, while via sequential blocks, only a single contained handler is allowed. Thus, to visually express any possible event associations, blocks must be nested, as depicted under Figure 10 (left part). However, as also shown in Figure 10 (right part), semantically the nested structures imply scoped event handlers, something not always true in underlying source code domain.

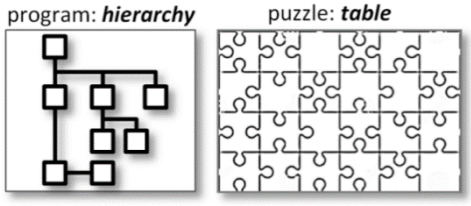

Figure 12: Potential mismatch between the jigsaw-puzzle metaphor and the hierarchical program structure.

The same scenario in a flow-based paradigm is represented with connected event handlers, but without implying nested scopes, as shown under Figure 11. Thus, for certain programming scenarios, it is clear that some visual metaphors are more appropriate than others. Interestingly, as outlined under Figure 12, while programs are genuine hierarchical structures, jig-saw puzzles have a two-dimensional tabular structure, without a hierarchy. In this sense, the *jigsaw metaphor* is either a mismatch for the target domain, or else and more precisely it is not used as in the real world, but only as a choice of *visual style* or abstract naming. One of the most notable cases is the flow-based paradigm, since, based on the original definition in (Morisson, 1994), is more a programming model than a visual metaphor. In fact, as outlined under Figure 13, historically there have been many flow-based visual styles, like flowcharts, processes (far before BPMN), and dataflow diagrams. As a real world analogy, the electric circuit constitutes the physical metaphor for all flow-based programming approaches.

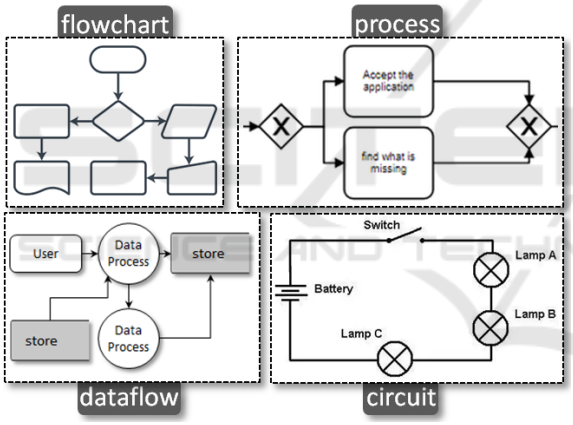

Figure 13: Flow-based metaphor with varying forms, with the electric circuit being the original physical metaphor.

As mentioned earlier, there is no silver bullet in choosing a single metaphor for visual programming, even when it seems to be particularly suitable for a target domain. Visual metaphors are abstractions on top of programming languages and models, meaning they hardly match all language constructs and the host application domain functionality. In this context, progress is needed to allow mixing metaphors together, likely with the interoperation of various visual editors, enabling switching to the most appropriate metaphor per case.

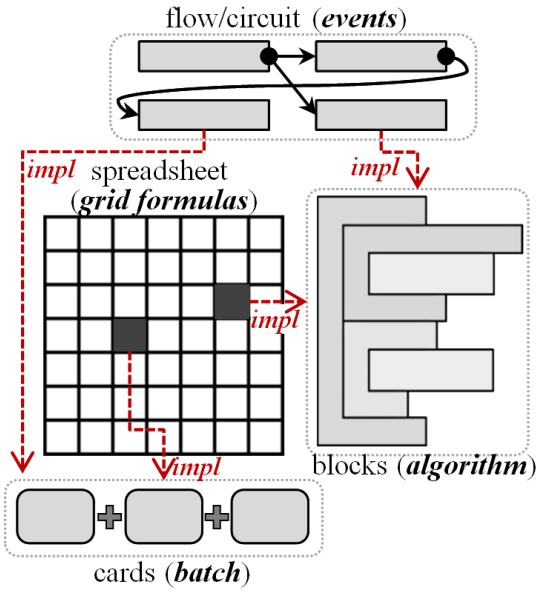

Figure 14: Combining metaphors enables adopting the most suitable paradigm in a given programming context.

As depicted under Figure 14, when implementing various parts of a single visual program, users may be able to switch between alternative forms and representations, supported by various connected and cooperating editors. Multi-paradigm and multi-metaphor visual environments reflect in a more integrated manner the varying semantic challenges of programming.

4.2 Reasonable Visual Complexity

When program size and complexity scales up, the layout policy should guarantee learners are enabled to manage the program in a *reasonable* manner. Clearly, programs may become quite large and complicated, making the overall programming task an inherently difficult and demanding activity.

In this context, the graphical representation of elements plays a critical role on the overall complexity, not only in terms of the overhead in perception and understanding, but also on how easy or difficult it is to extend and modify even moderately large programs. Existing visual systems significantly vary against this criterion.

As shown under Figure 15, flow-based structures tend to explode in visual complexity exponentially as the number of processes and connections tends to linearly increase. Effectively, programmers may lose

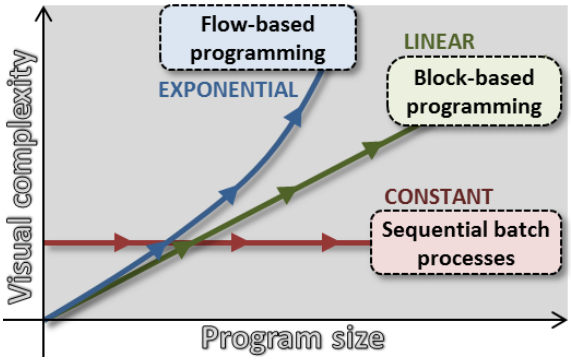

Figure 15: Visual complexity increase in relationship to program size for the three most popular paradigms.

control of the program and the component associations very easily, unless there are extra facilities helping them to organize large flow-based networks in a divide and conquer fashion.

On the opposite side sit *batch processes*, like Kudo game cards, possessing constant complexity, when streams of sequential operations grow with new entries. Finally, block-based visual systems fall somewhere in the middle in terms of visual complexity, since complexity mostly grows linearly with the number of involved program blocks.

4.3 Interactive Level of Detail

Interactively configurable level-of-detail enables to control which program parts remain visible during editing. Such facilities rely on structural, syntactic or semantic grouping, and may enable learners to switch on-the-fly between various views, hiding temporarily details that are unnecessary for the current editing task. In Figure 16, we depict folding and unfolding options in Scratch, helping reduce the overall visually occupied area of a program.

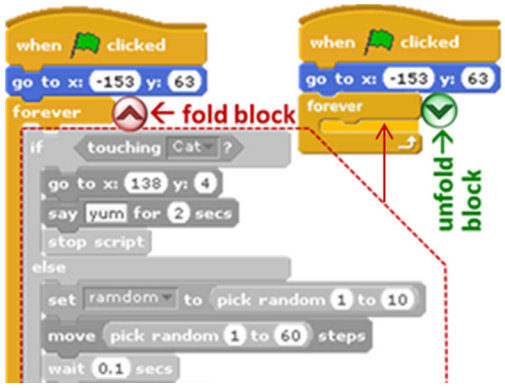

Figure 16: Mockup scenario showing folding / unfolding in block-based editors (here illustrated for Scratch).

Similar semantic folding may apply to flow-based editors as well, enabling to selectively display: (i) only the event sources of a certain type or category, like network messages or critical sensor notifications, (ii) consumers of a particular category such as widgets; and (iii) all items preceding or following in the flow a particular process block.

4.4 Tagged Annotations and Notes

Code annotations are user-defined comments, such as hints for corrections or improvements. In source code, they are usually inserted as text inside comments, mixed with code, while prefixed with some tags users may easily recognize and recall, like *TODO* and *FIXME*. Some tools like App Inventor (Wolber et al., 2014), allow comments on blocks in the form of editable callouts, but tagging is not semantically supported. An example mockup for introducing user-defined tagged notes in the App Inventor editor is outlined under Figure 17.

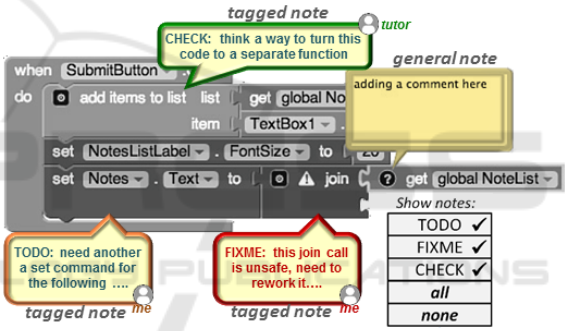

Figure 17: Tagged notes mockup for App Inventor, with display control and author info (user icon with title).

5 TOOL REQUIREMENTS

5.1 Project Management

Such features allow organize the programming work into distinct units or modules and may include extra facilities, like domain-specific instrumentation that may assists learners to setup a new program solution. For example, for small-scale games tools for terrain editing and asset authoring may be offered (Savidis & Katsarakis, 2021), while in the Internet of Things, management of smart device ecosystems might be provided (Savidis et al., 2021).

5.2 Debugging Facilities

Although visual programming systems are mostly targeted in learning where error resolution is crucial, visual debugging remains an underexplored territory. Such facilities should operate in alignment with the underlying language paradigm and the visual metaphor and layout, with program tracing working at the level of visual program units. For example, the block-level debugger for Blockly presented in (Savidis & Savaki, 2019) offers block-level tracing, with step commands operating syntactically at the level of distinct blocks. Finally, advanced features useful in learning may include: reverse execution and why lines (Myers et al., 2017).

5.3 Programming Assistants

Such tools are known as wizards and may guide learners in the process of visual coding. Their design should match the visual metaphor, but semantically they are more coupled with the underlying language paradigm. Thus, general-purpose tools may be developed working with editors of the same underlying programming model. Interactively, they may include coding templates, Q&A sessions, recommendations and procedural guidance (how-to). Syntax-directed editors (Agapakis, 2021) with descriptive tooltips and semantic help may also play a role similar to live coding assistants.

5.4 Custom Static Analysers

Since the programs in a learning context are typically small, such analysers should emphasize improvements and educational recommendations. For example, an analyser may suggest equivalent but simplified loop versions, propose the conversion of a code fragment to a function, or offer an alternative more readable and clean way to write a particular logical or arithmetic expression.

6 CONCLUSIONS

Visual programming systems are currently the primary instruments for the early teaching of basic programming skills, while they are increasingly deployed in various domains for rapid development by non-professional programmers. Compared to tools for professional programming, there are many functionality layers and features that can be introduced to improve the programming experience and better support the overall learning process.

In this paper, we presented a brief but systematic account of key design requirements for future visual development systems, relying on the new notion of programming experience, while having a primary learning orientation. Overall, we believe that such requirements can be more effectively addressed separately, by cooperating tools, within open and extensible future visual development environments.

REFERENCES

Agapakis, E. (2021). *Code-Chips: Interactive Syntax in Visual Programming*. Master Thesis, CSD, University of Crete, DOI: 10.13140/RG.2.2.28297.72801

Ball, T., Protzenko, J., Bishop, J., Moskal, M., Halleux, J., Braun, M., Hodges, S., Riley, C. (2016). *Microsoft touch develop and the BBC micro: bit*. In proceedings of ICSE 2016 (Companion), ACM, 637-640

E. Pasternak, R. Fenichel and A. N. Marshall (2017). *Tips for creating a block language with blockly*. IEEE 2017 Blocks and Beyond Workshop, 21-24.

Kiper, J., Howard, E., Ames, C. (1997). *Criteria for Evaluation of Visual Programming Languages*. IN Elsevier Journal of Visual Languages & Computing, Volume 8, Issue 2, 175-192.

Ko, R., Lee, S., Lee, E. (2009). *Business Process Management (BPM) Standards: A Survey*. In Business Process Management Journal, Emerald Group Publishing Limited. Volume 15 Issue 5.

Law, E., Roto, V., Hassenzahl, M., Vermeeren, A., Kort, J. (2009. *Understanding, Scoping and Defining User Experience: A Survey Approach*. Proceedings of CHI'09 Conference on Human Factors in Computing Systems. Boston, MA (4–9 April), ACM, 719–728

Maloney, J, Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010). *The scratch programming language and environment*. ACM Transactions on Computing Education (TOCE) 10.4, 1-15

MacLaurin, M. (2011). *The design of kodu: a tiny visual programming language for children on the Xbox 360*. SIGPLAN Notices, 46(1), ACM, 241–246

McGuffin, M., Fuhrman, C. (2020). *Categories and Completeness of Visual Programming and Direct Manipulation*. AVI 2020 (September 2020), ACM, Article No 7, 1–8

Morisson, J. P. (1994). *Flow-based Programming: A New Approach to Application Development*, Van Nostrand Reinhold.

Myers, B., Ko, A., Scaffidi, C., Oney, S., Yoon, Y., Chang, L. S-P., Kery, M.B., Jia-Jun, T. (2017). *Making End User Development More Natural*. New Perspectives in End-User Development 2017: 1-22

Myers, Brad. (1990). *Taxonomies of visual programming and program visualization*. Journal of Visual Languages & Computing 1.1 (1990), 97-123.

OpenJS Foundation. (2022). *Node-RED: Low-code programming for event-driven applications*.

https:// nodered.org/. Accessed online January 2022.

Repenning, A. (2017). *Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets*. In Journal of Visual Languages and Sentient Systems (JVLC) Vol 3 (July 2017), 68–91

Savidis, A., Savaki, C. (2019). *Complete Block-Level Visual Debugger for Blockly*. IHSED 2019, 286-292

Savidis, A., Katsarakis, A. (2021). *Game Development as a Serious Game with Live-Programming and Time-Travel Mechanics*. ICEC 2021, 181-195

Savidis, A., Valsamakis, Y., Linaritis, D. (2021). *Blockly Toolbox for Visual Programming of Smart IoT Automations*. IsAmI 2021, Salamanca, Spain (6-8 October), Springer LNNS.

Tawfik, A., Gatewood, J., Gish-Lieberman, J., Hampton A. (2021). *Toward a Definition of Learning Experience Design*. Technology, Knowledge and Learning. Springer. https://doi.org/10.1007/s10758-020-09482-2

Vallance, M., Martin, S., Wiz, C., Schaik, P. (2009). *LEGO Mindstorms™ for informed metrics in virtual worlds*. British HCI 2009, Cambridge, UK (1-5 September), ACM, 159-162

Wolber, D., Abelson, H., Friedman, M. (2014) *Democratizing computing with App Inventor*. Mobile Computing & Communications Review, 18(4) 53-58.