Graphical Temporal Structured Programming for Interactive Music

ABSTRACT

The development and authoring of interactive music or applications, such as user interfaces for arts & exhibitions has traditionally been done with tools that pertain to two broad metaphors. Cue-based environments work by making groups of parameters and sending them to remote devices, while more interactive applications are generally written in generic art-oriented programming environments, such as Max/MSP, Processing or openFrameworks. In this paper, we present the current version of the i-score sequencer. It is an extensive graphical software that bridges the gap between time-based, logic-based and flow-based interactive application authoring tools. Built upon a few simple and novel primitives that give to the composer the expressive power of structured programming, i-score provides a time line adapted to the notation of parameter-oriented interactive music, and allows temporal scripting using JavaScript. We present the usage of these primitives, as well as an i-score example of work inspired from music based on polyvalent structure.

1. INTRODUCTION

This paper outlines the new capabilities in the current iteration of i-score, a free and open-source interactive scoring sequencer. It is targeted towards the composition of scores that will have an interactivity component, that is, that are meant to be performed to some extent while maintaining an ordering or structure of the work either at the micro, macro, or both levels. While it can be used for music composition, it is not restricted to solely auditory arts but can instead control any kind of multi-media work.

We first expose briefly the main ideas behind interactive scores, and explain how i-score can now be used as a language of the structured programming language family, targeted towards temporal compositions, in a visual time-line interface.

In previous research[1, 2] interactive triggers were exhibited as a tool for a musician to interact with the computer following a pre-established score. Other approaches for real-time interactive scoring are for instance seen with Antescofo[3] which allows for live score following. Here, we show that with the introduction of loops, and the capacity to perform computations on variables in a score, we are able to use interactive triggers as a powerful flow control tool, which allows to express event-driven constructs, and to build a notion similar to traditional programming languages procedures.

We conclude by exhibiting an i-score example of a musical work inspired by polyvalent structure music, that can be used by composers as a starting point to work with the environment. This example contains relatively few elements, which shows the practical expressiveness of the language.

2. EXISTING WORKS

The sequencer metaphor is well-known amongst audio engineers and music composers. It is generally composed of tracks, which contains audio or MIDI clips, applied effects and parameter automations.

In multiple cases, it has been shown that it was possible to write more generalist multimedia time-line based sequencers, without the need to restrict oneself to audio data types. The MET++ framework[4] is an object-oriented framework tailored to build such multimedia applications. A common approach, also used in previous version of i-score, is to use constraint programming to represent relations between temporal objects[5, 2, 6]. This is inspired from Allen's relationship between temporal objects. In [7], Hirzalla shows how conditionality can be introduced between multimedia elements in a time-line to produce different outcomes.

Other approaches for interactive music are generally not based on the time-line metaphor, but more on interaction-centric applications written in patchers such Max/MSP or PureData, with an added possibility of scoring using cues. Cues are a set of parameters that are to be applied all at once, to put the application or hardware in a new state. For instance, in a single cue, the volume of a synthesizer may be fixed at the maximum value, and the lights would be shut off. However, the temporal order is then not apparent from the visual representation of the program, unless the composer takes care of maintaining it in his patch. When using text-based programming environments, such as Processing or OpenFrameworks, this may not be possible if concurrent processes must occur. For instance, having a sound playing while the lights fade-in.

The syntax and graphical elements used in i-score as well as the execution semantics are for the most part introduced in [8, 9]. Multiple execution semantics for the same graphical formalism have been developed. They are based on Petri Nets[10], Time Automata[11], reactive languages[12] and temporal concurrent constraint programming[13].

The novelty of our approach lies in the introduction of graphical temporal loops, and of a computation model based on JavaScript that can be used at any point in the score. These two features, when combined, provide more expressive power to the i-score visual language, which allows for more dynamic scores.

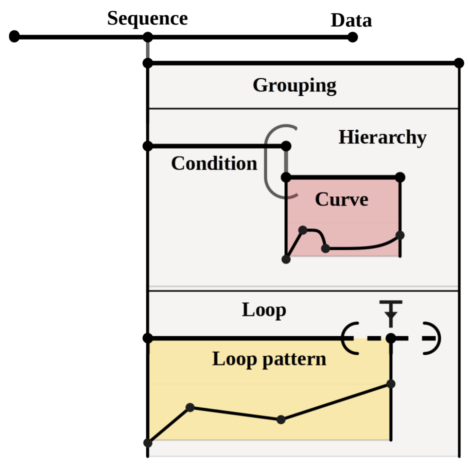

Figure 1. Screen-shot of a part of i-score, showing major elements of the formalism. The *time constraint* is the full or dashed horizontal line, the *states* are the black dots, the *time nodes* are the vertical bars, and a *time event* is shown at the right of the "Condition" text. Interactive *triggers* are black **T**'s with a downwards arrow. There are five *time processes* (capitalized) : a *scenario* which is the hierarchical root of the score, another *scenario*, in the box "Hierarchy", an *automation* on a remote parameter in the "Curve" box, a *loop* in the box containing the *loop* pattern, and another *automation* that will be looped.

3. TEMPORAL STRUCTURED PROGRAMMING

Structured programming is a programming paradigm which traces back to the 1960's, and was conceived at a time where the use of `GOTO` instructions was prevalent in programming, which often led to code difficult to understand.

The structured programming theorem[14, 15] states that any computable function can be computed without the use of `GOTO` instructions, if instead the following operations are available :

- Sequence (`A` followed by `B`),
- Conditional (`if(P) then A else B`),
- Iterative (`while(P) do A`).

Where `P` is a boolean predicate, and `A`, `B` are basic blocks. Additionally, the ability to perform computations is required in order to have a meaningful program.

In order to allow for interactive musical scores authoring, we introduce these concepts in the time-line paradigm. A virtual machine ticks a timer and makes the time flow in the score graph. During this time, multiple processes, including hierarchical scoring, looping, and multimedia processes, are run at each tick.

Processes can be of two kinds : temporal or instantaneous. Temporal processes can be explained as functions of time that the composer wants to run between two points in time. For instance : *do a volume fade-in from t=10s to t=25s*.
Instantaneous processes are functions that will be run at a single point in time. For instance : *play a random note*.

3.1 Scenario

The *scenario* is a particular setup of the elements of the i-score model : *time constraint* (a span of time which contains temporal processes), *time node* (which will synchronize the ending of *time constraints* with the happening of an external event such as a note being played), *time event* (a condition that will cause its following *time constraints* to start if it is true), and *state* (which contains data to send and instantaneous processes). They are shown in fig. 1. The time flows from left to right as in most traditional sequencers. However, due to the presence of interactivity, the various possibilities of execution of the score cannot be shown. The presence of a single interaction point makes the state space enormous. Hence dashes are shown when the actual execution time is not known beforehand. For instance : *play a D minor chord until a dancer moves on stage*.

In the context of a *scenario*, as shown in [8], these primitives allow for sequencing elements, conditional branching, and interactive triggering, but are not enough for looping. The user interface of the i-score software allows for all the common and expected operations when editing a *scenario*: displacement, scaling, creation, deletion, copy-paste...

3.2 Loop

The loop is another setup of the same elements, more restrictive, and with a different execution algorithm : it is composed exclusively of two *time nodes*, two *time events*, two *states*, and a *time constraint* in-between, which we name loop pattern. When the second *time node* gets executed, the time flow is reverted to before the execution of the first *time node*. This means that if the composer adds an interactive trigger on any of these *time nodes*, the *loop* will be able to play patterns of different durations at each loop cycle, with a different outcome. This is strictly more general than loops in more traditional sequencers such as Avid Pro Tools, Steinberg Cubase or Apple Logic, when looping consists in a strict duplication of the audio or MIDI data.

As for the *scenario*, the *loop* is implemented as a temporal process that will be inserted under an existing *time constraint*. Hence the *loop*'s overall duration will be limited by its parent constraint's duration.

3.3 Communication

The software mainly communicates via the OSC[1] protocol. It maintains a tree of parameters able to mirror the object model of remote software built with Max/MSP, PureData, Unity3D or any OSC-compliant environment. An OSC-based RPC[2] protocol, Minuit, able to handle automatic discovery, can also be used. Conditions, interactive triggers, and more generally all processes of i-score have unrestricted access to this tree. They can read its data, perform computations on it, and send it back to the remote software or hardware. In the course of this paper, "device tree" refers to this tree.

Musical information can currently be expressed with the help of either a remote software or hardware device able to convert OSC messages to either MIDI or audio. One can for instance easily build a PureData patch to convert OSC messages matching a certain pattern to the corresponding MIDI notes.

[1] Open Sound Control
[2] Remote Procedure Call

3.4 Variables

Having variables is important : a variable can hold data, and a computation can be performed on it. For instance : $x \leftarrow x + 1$ is a simple incrementation operation, which requires one variable x. Different programming languages have distinct takes on the capabilities of variables. For instance, pure functional programming languages will generally behave according to immutability rules, that is, a variable once assigned cannot be modified. This is in contrast with imperative languages, closer to the machine's memory model with addressable and mutable data.

The implementation of variables is based on the device tree, which acts like a global memory. The composer can create new variables graphically in the tree, and select their types. The variables are statically typed:

- A variable's type must be chosen between integer, boolean, floating point, impulse, string, character, or tuple.
- Standard comparison operations can take place : for instance an integer and a floating point number will be able to be compared.

While there is no mechanism to allow dynamic allocation of new variables during the execution of a score, it could easily be implemented by having a special address. This address would take in argument the name of the variable to allocate as well as its type and add it as a node of the tree.

Finally, there is no notion of scope in the device tree : any process can access to any variable, which should be considered elements of the object model of the local software. However, when local variables are necessary for complex computations, it may be better to use the embedded scripting facilities to encapsulate them.

4. AUTHORING FEATURES

i-score provides multiple authoring features aiming to give a fair amount of freedom to the composer. Besides, the software is based on a plug-in architecture which allows for easy extension of its capabilities. Here is a list of the most prominent recent additions to i-score :

- JavaScript support : one can use JavaScript scripts both as temporal and instantaneous processes. When writing a temporal process, the composer has to provide a function of the following form :

  ```
  function(t) {
      var obj = new Object;
      obj.address = '/an/address';
      obj.value = 42 * t;
      return [ obj ];
  }
  ```

  This function will get called at each tick with the time `t` being a floating point between 0 and infinity, 1 being the default date that the composer has set. It will produce a linear ramp on the `/an/address` parameter, that will go from zero to 42 during the duration of the parent *time constraint* of this process. If the process lasts longer than its expected duration due to interactive triggering, `t` will become greater than one. The messages returned will be applied to the local state of the tree and sent remotely if corresponding nodes were found. When writing an instantaneous process, the function to provide is similar, but does not take a time argument. Functions writers should be encouraged to write stateless functions, since it would allow for optimizations opportunities such as precomputing an array of values. However, it is not possible to prove that an arbitrary JavaScript code will not modify external context. Finally, a global object provides an API to query the current state of other parameters in the device tree. This allows for arbitrarily complex mappings between parameters.

- Automation : the traditional DAW [3] automation, which writes in a parameter over time. Currently provided are 1D (linear, power) and 3D (cubic spline) automations.
- Mapping : similar in appearance to the automation, this process will take an OSC address as input, apply a transfer function drawn graphically, and write the output to another address.
- Recording : one can record either automations which will be able to apply automatic interpolation for numeric parameters, or record any kind of input message to replay it as it happened afterwards.
- Execution speed control : the temporal constraints all have a multiplicative factor for execution speed.
- Introspectability : i-score exposes the objects of the current score in the same tree that the remote devices. It follows the hierarchical model defined by the *scenario* and the *loop*. Multiple attributes can be queried, and to some extent modified during the execution of the score. For instance, the activation status of interactive triggers, and the execution speed of all the temporal constraints are controllable parameters. The elements are organized hierarchically according to their hierarchy in the *scenario*, and can also be remote-controlled from the network. Plugins can expose their own attributes in this tree.
- Interactive execution : during the authoring process, it is sometimes necessary to replay just a part of the score. i-score provides this feature. However, it is necessary for the composer to choose beforehand the truth value of the conditions he wants to test. There are two "debug" execution modes : one that will directly start playing from where the user points the mouse, and another that will try to merge all the previously sent values, to put the external environment in the same state that it would be if the score had been played normally up until this point. This would however not work if there are physical processes that are not instantaneous, as would be the case for smoke machines for instance.

[3] Digital Audio Workstation

5. TEMPORAL DESIGN PATTERNS

In this section, we present two design patterns that can be used for writing an interactive score. We will first showcase event-driven scores, which can be used to get a behaviour similar to a traditional computer program executing instructions in sequence without delay, or common network communication tasks. Then, we will present an example of the concept of procedure in a time-oriented model.

5.1 Event-driven design

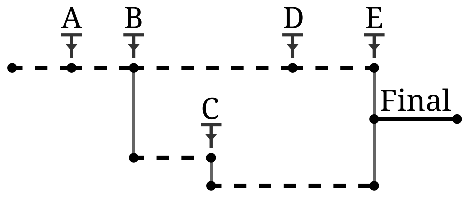

Figure 2. An example of event-driven score : if all the interactive trigger's conditions are set to true, they will trigger at each tick one after the other. Else, standard network behaviour is to be expected.

Event-driven, or asynchronous design is a software design paradigm that is centered on the notion of asynchronous communication between different parts of the software. This is commonly used when doing networked operations or user interfaces, since they have to be responsive.

```
boolean c_was_played := false
boolean d_was_played := false

on A.triggered :
 on B.triggered :
  on C.triggered :
   c_was_played := true
   if d_was_played :
    send E.canWait

  on D.triggered :
   d_was_played := true
   if c_was_played :
    send E.canWait

  on E.canWait :
   on E.triggered :
    play Final
```

Figure 3. The fig. 2 score in a callback-based event-driven programming style.

In textual event-driven programming, one would write a software using callbacks, futures or reactive programming patterns[16]. An example of what could be the score of fig. 2 in a textual language is given in fig. 3.

Since i-score supports interactive events, one can easily write such event chaining. An example is given in fig. 2. The advantage is that ordered operations can be written easily : there is no possibility for B to happen before A if there is a *time constraint* between A and B. However, the execution engine will introduce a delay of one tick between each call. The tick frequency can be set to as high as one kilo-hertz. Synchronization is achieved purely graphically : here, the last *time constraint*, in solid black, will only be executed after all the incoming branches were executed. This allows to write a score such as : *start section B five seconds after musician 1 and 3 have stopped playing*. There is no practical limit to the amount of branches that can be synchronized in this way.

5.2 Simulating procedures

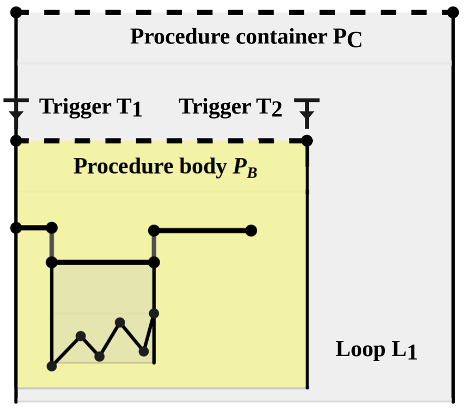

Figure 4. Implementation of a procedure in i-score

The notion of procedure is common in imperative programming languages. It consists in an abstraction around a behaviour that can be called by name easily. It can be extended to the notion of function which takes arguments and returns values. Procedures are useful: if one wants the same behaviour to occur at multiple times during the execution of the program or the score, this behaviour will have to be written only once. However, the visual flow becomes deconstructed, because the definition of the procedure is given at a point in the program or score, and its usage occurs at other points.

We present in fig. 4 how one could create a simple procedure P that will then be able to be recalled at any point in time. However, it suffers from a restriction due to the temporal nature of the system : it will only be able to be called when it is not already currently running. This is due to the single-threaded nature of the execution engine : there is only a single playhead for the entirety of the score, hence a single temporal process cannot be processed at the same time at two different points.

The procedure is built as follows :

- A *time constraint*, P_C in the root *scenario* will end on an interactive triggering set with infinite duration.

- This *time constraint* contains a *loop* L_1. The procedure is named `p` in the local tree. The interactive triggers T_1, T_2 at the beginning and end of the pattern *time constraint* are set as follows :

  - T_1 : `/p/call true`.
  - T_2 : `/p/call true`.

  A *state* triggered by T_1 should set the message: `/p/call false`. This causes the procedure not to loop indefinitely : it will have to be triggered manually again.

- The *loop*'s pattern P_B contains the actual procedure data, that is, the process that the composer wants to be able to call from any point in his score.

The execution of this process will then overlay itself with what is currently playing when at another point of the score, the message `/p/call true` is sent. Once the procedure's execution is finished, it enters a waiting state until it is called again. This behaviour is adapted to interactive arts: generally, one will want to start multiple concurrent processes (one to manage the sound, one to manage videos, one to manage lights…) at a single point in time; this method allows to implement this.

6. MUSICAL EXAMPLE : POLYVALENT STRUCTURE

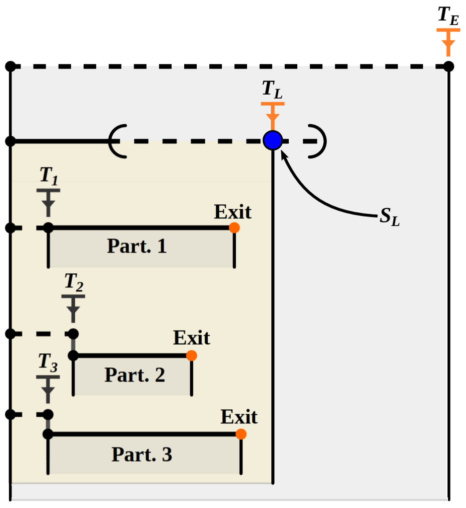

Figure 5. A polyvalent score in i-score

In this example, showcased in fig. 5, we present a work that is similar in structure to Karlheinz Stockhausen's *Klavierstück XI* (1956), or John Cage's *Two* (1987).

The complete work contains variables in the tree, as explained earlier, and a temporal score. The tree is defined in fig 6.

Address	**Type**	**Initial value**
`/part/next`	integer	chosen by the composer
`/part/1/count`	integer	0
`/part/2/count`	integer	0
`/part/3/count`	integer	0
`/exit`	boolean	false

Figure 6. Tree used for the polyvalent score

For instance, `/part/next` is an address of integral type, with a default value chosen by the composer between 1, 2, 3 : it will be the first played part. The score is as follows : there are multiple musical parts containing recordings of MIDI notes : **Part. 1, 2, 3**. These parts are contained in a *scenario*, itself contained in a *loop* that will run indefinitely. At the end of each part, there is an orange *state* that will write a message "true" to a variable `/exit`. The pattern of the *loop* ends on an orange interactive trigger, T_L. The *loop* itself is inside a *time constraint* ended by an interactive trigger, T_E. Finally, the parts are started by interactive triggers $T_{\{1,2,3\}}$.

The conditions in the triggers are as follows :

- $T_{\{1,2,3\}}$:
  `/part/next == {1, 2, 3}.`
- T_L :
  `/exit == true.`
- T_E :
  `/part/1/count > 2 or`
  `/part/2/count > 2 or`
  `/part/3/count > 2.`

The software contains graphical editors to set conditions easily. Finally, the blue *state* under T_L contains a JavaScript function that will draw a random number between 1 and 3, increment the count of the relevant `/part`, and write the drawn part in `/part/next`. This function is shown in fig. 7. If any count becomes greater than two, then the trigger T_E will stop the execution : the score has ended. Else, a new loop iteration is started, and either T_1, T_2 or T_3 will start instantaneously.

Hence we show how a somewhat complex score logic can be implemented with few syntax elements.

Another alternative, instead of putting MIDI data in the score, which makes it entirely automatic and non-interactive, would be to control a screen that displays the part that is to be played. A musician would then have to interpret the part in real-time which would bring back the human aspect.

```
function() {
  var n =
    Math.round(
      Math.random()*2)+1;

  var root = 'local:/part/'
  return [ {
    address : root + 'next',
    value   : n
  }, {
    address : root + n,
    value : iscore.value(root + n) + 1
  } ];
}
```

Figure 7. JavaScript code contained in the end *state* in the example of fig. 5.

7. TOOLING

There are plenty of parts that requires new problems to be solved in i-score, especially in the area of debugging. For instance, debugging is generally used to understand and find out easily why did not a program produce the expected result. In i-score, there is no low-level memory management which removes a large class of errors. However, as soon as the song's logic becomes complex, it is useful to have tools that give the authors the possibility to pinpoint where the execution of the show does not match the expected behaviour. Since the presented formalism has extensive expressive capabilities, the composer might find himself confronted to the same family of problems that programmers encounter daily. Nowadays, many programming language offer very robust tooling, as can be seen in the performance tools that are provided in integrated development environments such as Eclipse, Xcode, Visual Studio, and the Unix development tools[17]. Such tooling can decrease the development time a lot, and we argue that its application in interactive scores would lead composers to write author more easily and efficiently. Already mentioned is the capacity to play from any point in time in i-score. Other possibilities would be to preview what a set of events would lead to during execution, debug logs, and backwards execution.

8. CONCLUSION

We presented in this paper the current evolutions of the i-score model and software, which introduces the ability to write interactive and variable loops in a time-line, and the usage of JavaScript to perform arbitrary computations on the state of the local and external data controlled by i-score.

Currently, the JavaScript scripts have to be written in code, even if it is in a generally visual user interface. But given enough testing and user evaluation, it could be possible to have pre-built script presets that could be embedded in the score for the tasks that are the most common when writing a score.

Additionally, we aim to introduce audio and MIDI capabilities in i-score, so that it will be able to work independently of other sequencers. For instance, should it play a sequence of three sounds separated by silence, it would be difficult for the composer if he had to load the songs in an environment such as Ableton Live, and work with them remotely from the other time-line of i-score.

This would also allow for more control on the synchronization of sounds : if they are controlled by network, the latency can cause audio clips that are meant to be synchronized in a sample-accurate manner to be separated by a few milliseconds, it is enough to prevent the usage in some musical contexts.

Acknowledgments

Anonymized

9. REFERENCES

[1] T. De la Hogue, P. Baltazar, M. Desainte-Catherine, J. Chao, and C. Bossut, "OSSIA : Open Scenario System for Interactive Applications." Journées de l'Informatique Musicale, 2014.

[2] A. Allombert, G. Assayag, M. Desainte-Catherine, C. Rueda *et al.*, "Concurrent constraints models for interactive scores," in *Proc. SMC 2006*, 2006.

[3] A. Cont, "ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music." in *International Computer Music Conference (ICMC)*, 2008, pp. 33–40.

[4] P. Ackermann, "Direct manipulation of temporal structures in a multimedia application framework," in *Proceedings of the second ACM international conference on Multimedia*. ACM, 1994, pp. 51–58.

[5] J. Song, G. Ramalingam, R. Miller, and B.-K. Yi, "Interactive authoring of multimedia documents in a constraint-based authoring system," *Multimedia Systems*, vol. 7, no. 5, pp. 424–437, 1999.

[6] M. Toro-Bermúdez, M. Desainte-Catherine *et al.*, "Concurrent constraints conditional-branching timed interactive scores," in *Proc. SMC 2010*. Citeseer, 2010.

[7] N. Hirzalla, B. Falchuk, and A. Karmouch, "A temporal model for interactive multimedia scenarios," *IEEE MultiMedia*, no. 3, pp. 24–31, 1995.

[8] J.-M. Celerier, P. Baltazar, C. Bossut, N. Vuaille, J.-M. Couturier *et al.*, "OSSIA: Towards a unified interface for scoring time and interaction," in *Proceedings of the 2015 TENOR Conference*, 2015.

[9] P. Baltazar, T. de la Hogue, and M. Desainte-Catherine, "i-score, an Interactive Sequencer for the Intermedia Arts," pp. 1826–1829, 2014.

[10] A. Allombert, G. Assayag, M. Desainte-Catherine *et al.*, "A system of interactive scores based on Petri nets," in *Proc. SMC 2007*, 2007, pp. 158–165.

[11] J. Arias, M. Desainte-Catherine, and C. Rueda, "Exploiting Parallelism in FPGAs for the Real-Time Interpretation of Interactive Multimedia Scores," in *Proceedings of the Journées d'Informatique Musciale 2015*, 2015.

[12] J. Arias, M. Desainte-Catherine, S. Salvati, and C. Rueda, "Executing Hierarchical Interactive Scores in ReactiveML," *Actes des Journées d'Informatique Musicale*, pp. 25–34, 2014.

[13] M. Toro, M. Desainte-Catherine, P. Baltazar *et al.*, "A model for interactive scores with temporal constraints and conditional branching," *Proc. JIM 2010*, pp. 31–38, 2010.

[14] C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, May 1966.

[15] H. D. Mills, "Mathematical foundations for structured programming," 1972.

[16] K. Kambona, E. G. Boix, and W. De Meuter, "An evaluation of reactive programming and promises for structuring collaborative web applications," in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. ACM, 2013, p. 3.

[17] D. Spinellis and S. Androutsellis-Theotokis, "Software Development Tooling: Information, Opinion, Guidelines, and Tools," *IEEE Software*, no. 6, pp. 21–23, 2014.