Computational Thinking Tools:

Analyzing concurrency and its representations

Cleyton Slaviero
Departamento de Informática (DI)
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rio de Janeiro, RJ - Brazil
cslaviero@inf.puc-rio.br

Edward Hermann Haeusler
Departamento de Informática (DI)
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rio de Janeiro, RJ - Brazil
hermann@inf.puc-rio.br

***Abstract*— Computational thinking (CT) tools, as a software system, express their designers' perspective on how a selected set of Computer Science concepts should be introduced, typically hiding details to avoid unnecessary complexity. This paper focuses on how concurrency is dealt with by five well-known tools in this domain: Scratch, Alice, AgentSheets, NetLogo and Greenfoot. We present the results of a systematic analysis contrasting their model of concurrent behavior with the corresponding metamessages, the messages about messages of concurrency, that trigger users' interpretation and learning of concurrency-related concepts. We present and discuss the conceptualizations that potentially emerge from using these five tools and compare them with established concurrency concepts. Our findings indicate opportunities for an explicit exploration of how some concurrency aspects are implemented in games and simulations built with CT tools. We believe that this might facilitate future learning and comprehension of complex concurrency concepts, considering that the knowledge embedded in these tools can also influence students' understanding of concurrency.**

Keywords*— *computer science education, programming, concurrent programming, semiotic engineering

I. INTRODUCTION

From the early days of the LOGO programming language, created by Papert [11] as an educational tool for introducing Computer Science (CS) concepts to young children, many other tools have been developed to support computational thinking acquisition (CTA) over the years and with many distinct goals on mind. These tools are of a special kind of programs: they implement the very same concepts that will be taught by using them. For designers of these tools, it then becomes a challenge to balance complexity of the implemented CS concepts with the ease of use while enabling users to have simple, yet powerful tools to express themselves through their creations. If designers present all the complexity to users, these tools may be too hard for novice programmers to understand; if too easy, students may find it boring [13].

For engaging students in CTA, many tools focus on the creation of games and simulations by providing multiple resources (visual programming environments, domain-oriented languages, among others) for developing them. This strategy for teaching programming helps students to easily visualize the results of their implementations with fun and engagement [14]. When building these games and simulations, students must deal with multiple agents interacting with each other, manipulating variables, and other programming aspects. Some of these aspects relate closely to the concurrency domain, which studies how processes may run in parallel without negatively affecting each other [2]. Thus, these tools must implement concurrency concepts to properly run students' games and simulations, which adds to the set of concepts designers must consider when building these environments.

As literature reveals no tool is suited for every concurrent programming tasks [15]. MultiLogo, for instance, is well-suited for robotic tasks, given its matching of the programming language with a robot's movements, but it could be difficult to program massively concurrent processes, which would demand primitives to manipulate and orchestrate them, either in a centralized or decentralized manner [20]. Thus, when presenting concurrency to users, it is important to consider which aspects of concurrency need to be modeled. Otherwise, whether these students become professional programmers or not, these tools may impact students learning of CS concepts due to inconsistencies between the concepts they learned and the problems they might need to tackle. This could generate, for instance, weak connections about synchronization as they learned previously when trying to use the same concepts in other problems [3].

To this extent, our research question relies on if and how these trade-offs impact on how conceptualizations are presented to users of these CT tools. We begin to investigate this matter in this paper, in which we present a systematic evaluation to investigate if and how concurrency is conceptualized in five CT tools widely used to support CTA: AgentSheets[1], Scratch[2], Greenfoot[3], NetLogo[4] and Alice[5]. These tools aim at simplifying the programming process by

[1] http://www.agentsheets.org

[2] https://scratch.mit.edu

[3] http:// www.greenfoot.org

[4] https://ccl.northwestern.edu/netlogo/

[5] http://www.alice.org

providing environments simpler than IDEs to develop programming activities, focusing on novice programmers.

We implemented a modified version of the Dining Philosophers problem, well known in concurrency domain, as a case of concurrent behavior to be used as basis to discuss concurrency in these tools. By analyzing implementations, we programmed in each of the studied tools, we discovered three models of concurrent behavior. In the literature, Petri Nets is a well-known notation to represent concurrent behavior [12], and it allowed us to better understand how each tool models a concurrent behavior. Also, to investigate how concurrency depicted in these models are communicated by designers, we employed a semiotic analysis using the Semiotic Inspection Method (SIM), a method from Semiotic Engineering which focuses on the emission of designers' intent [18]. In our research, we focus on communication of concurrency in these tools. We conclude our analysis by providing a contrast between the models of concurrent behavior and results of the SIM's application. Finally, we discuss our findings and future research that could emerge from the results of this study.

II. Concurrent Programming and CT Tools

Concurrent programming is a Computer Science topic related to the study of processes in which resources must be shared, such as CPU time, variables, memory addresses, among others. It studies problems related to coordination, synchronization and communication of processes to share these resources maintaining consistency between each transition of the machine's state. As Ben-Ari defines, "A concurrent program is a set of sequential programs that can execute in parallel." [2]. Hoare also brings an interesting definition of concurrency [22]. It talks about interactions between processes that require participation from both at the same time, in the same event. The question is then how to precisely describe and manipulate these kinds of events.

There are many approaches for describing concurrent programming. One of the first attempts of formalization of the concept comes from Hoare [21], who described their concept of Communicating Sequential Processes (CSP). The Actor model is another kind of formalism, in which processes are seen as actors [22]. In both cases, they are communicating to each other to reach an understanding in the concurrent event. Petri Nets is another formalism for describing concurrent behavior. It focuses on analyzing concurrent events from the perspective of causality, which allows us to ignore, for the sake of clarity, physical restrictions [12].

For CS professionals, concurrency is becoming of a great importance, mainly due to the increase of multicore processors and environments, which demands that professionals consider a concurrent world when programming for it [5] However, authors agree that concurrency can be a challenging topic to teach [7]. The non-deterministic nature of concurrent programs, for instance, can confuse students unfamiliar with concurrent behavior, especially regarding the mastery of programming tools and understanding the underlying data structure that provides concurrent behavior [3]. Other authors explore the teaching of concurrency to students in the early years [5], which could bring interesting insights to the learning and a toolbox for more complex learning of concurrency later in the students' life. However, bringing a single model of concurrency is still not ideal, since it could make the student think that there is only one way to solve concurrency [15]. Thus, although it is important to teach concurrency, we also should make students and instructors aware of these different possibilities of solving concurrency tasks.

Researchers explore concurrency in many ways, often focusing on students' understanding of the concept [15], using the tools conceptualization of the concept and exploring it with students [5], and studying the understanding of specific concurrency concepts by students [1,3,7,16]. Although there are works employing CT tools to support the learning of concurrency, on Scratch for instance [10], little is mentioned about how the implemented concurrency influences on the learning activity. This happens mostly because the focus of these studies is on the concept being taught through the tool, and not by the tool. In this paper, we are interested in the second perspective, which we consider it can also influence the understanding of CS concepts, especially concurrency.

Also, as pointed out by Resnick, no single concurrency model is well suited for every situation [15]. By model of concurrency, we consider how these tools deal with concurrent behavior implemented by users, for instance using semaphores, locks, and other strategies to solve concurrency issues, and the outputs which follow these strategies. Due to students' creativity, implementation scenarios not well suited for the current CT tool often appears, which demands that instructors explain why its seemingly correct program is not running as expected when comparing his/her natural view of the world to a computational one [13,17], or why in the middle of the CT acquisition his/her hypothesis about this world starts to fail. These issues could demotivate them, when their recently acquired knowledge appears not to work anymore. Thus, it sounds reasonable to understand what the root of these problems could be: CT tools' conceptualizations about concurrency.

III. METHODOLOGY

In a broader context, this research aims at studying whether there is a systematic and effective way of knowing if and how CT tools used in pre-college education promote computer science conceptualizations which are consistent with what CS majors learn in college education. In this paper, our goal is to investigate the conceptualization of concurrency in a small set of CT tools. To this extent, we employed qualitative and exploratory research to investigate concurrency emerging from these tools.

Our research method encompasses 6 steps: two for defining the case to be studied, three related to the implementation and analysis of the execution of the generated programs, one for the Semiotic Inspection Method (SIM) [2], and one final step to contrast the perceived execution and the results of the semiotic analysis conducted using SIM.

Step 1. We chose a case to be studied and implemented a set of tools to be analyzed. Regarding the case, we chose a modified version of the Dining Philosophers, a well-known problem in concurrency domain [3]. In the original problem, five philosophers are seated around a round table and can either think or eat. To eat, one philosopher must grab two forks, one of each adjacent to each philosopher on its left and right sides, respectively. If the fork is not available, he must wait until it is, to eat. After eating, he puts back both forks on the table and starts thinking again. In our modified version, philosophers can

only eat after the simulation starts running, and once two forks are available, he grabs both forks, without releasing them. This allows us to study how each of the CT tools we selected deal with concurrency, instead of focusing on the problem itself.

Regarding the tools, we selected five widely used tools for games and simulation programming. These are AgentSheets, Scratch, Greenfoot, NetLogo and Alice. These tools were selected because they deal with agents' interaction, which is a kind of concurrent behavior, since these agents can either be processes and resources. In these environments, they interact to each other via other agents, their visual image, or using variables. These elements can be seen as shared resources, given the context they are used, which raise the same issues from the concurrency domain. Thus, these tools must implement some concurrency-related solutions, to run games and simulations which, as mentioned before, are often implicit to the user and thus motivated our research.

Step 2. In order to follow a consistent implementation process to compare each tool's implementation of the case studied, we defined a pseudo-code for the implementation process, as follows:

```
Define Philosophers from 1 to 5;
Define Forks from 1 to 5;
For each Philosopher
        Place it adjacent to philosopher i-1 and
    i+1;
For each Fork
    Place it between forks i-1 and i+1;
For each philosopher
        If left-fork and right fork are available,
    then get both forks;
For each Fork
        If fork is taken, lock it from being taken
    by any other Philosopher;
```

Step 3. Next, we proceeded with the implementation of the pseudo code from Step 2 in the CT tools. This led us to depict multiple variations of the programs in each tool, since a tool allows us to program a same case in many ways. As an example, on AgentSheets we could define the five philosophers as one unique Philosopher *agent* with five *depictions* (graphical representation of an agent), or five Philosophers, each with its own depiction. Implementing these variations was needed to analyze if there were variations to the execution between different representations of the same case in a given CT tool. We describe these distinctions in the implementations in section V.

Step 4. After implementing multiple versions of our simplified Dining Philosophers, we ran each implementation to uncover the model of concurrent behavior. For each CT tool, we analyzed each implemented version output to depict how agents were visited. From this, we elaborated models about the concurrent behavior. In order to being able to discuss them properly, we used Petri Net as a modeling notation [4]. In the literature, Petri Net is widely used to model and study concurrent behavior. It allows us to separate the problem from implementations, which is interesting if we would like to analyze the solutions implemented in different tools. A Petri Net is depicted as a directed graph consisting of places, transitions and tokens [4]. A place can be a state, a resource or a process, depending on the given semantic. A transition represents an action that can be performed by the system modeled. A token, put inside a place, is a marking that indicates the context of the net. Any transitions whose places entering in it contain tokens are called "enabled". A Petri Net changes its state when every enabled transition is fired, which makes the token "walk" from the input place to its output places, each one receiving one token. A Petri Net model is suited for describing non sequential behavior [5]. In this way, given two events "a" and "b", it is possible to distinguish an execution of "a" after "b" from an execution of "b" after "a", not by the time they occur, but by their causality relation. By doing this, we can model what is called truly concurrent events.

In our context, the Petri Net model can be employed as a bridge between the "natural world" and the computational world, in the sense that it allows to model the truly concurrent behavior that emerges from a problem, from a physical microworld, to these tools solutions, or the computational microworld [6] in order to reveal how concurrency issues are tackled by these tools.

Step 5. With these models of concurrency in hand, we proceeded to a careful semiotic inspection of these tools to understand how the models evidenced in step 4 are communicated via interface, along with how concurrency itself emerges from it. This semiotic analysis follows the Semiotic Inspection Method (SIM) [2], which allows us to investigate the emission of the designers' metacommunication about concurrency through these tools.

Semiotic Engineering characterizes Human-Computer Interaction as a shared communication between designers and users, in which designers, via metalinguistic, dynamic and static signs perform the communication of a message about the software, or metacommunication. These signs, which are anything representing something to someone [2], shape the metacommunication message. SIM is one of the methods of Semiotic Engineering and it helps us to explore the emission of the metacommunication message by designers at interaction time, via analysis of metalinguistic, static and dynamic signs. After thorough interpretation of these signs the researcher can reconstruct the designer's message, using the metacommunication template. This template summarizes the designers' perceptions and expectations about the users and their needs and expectation and provides the designer's characterization of what users should or must do to fulfill their intentions.

For this paper, we focused on the designers' message about concurrency in each CT tool we analyzed. To this extent, we considered a user, who already had knowledge about the tool and could create games and simulations but had never faced a concurrency issue. Then, his/her instructor asks him/her to build the modified version of the Dining Philosophers, which as depicted before, raises these issues for him/her to investigate. In this paper, we focus on the final metacommunication message of each tool and explore details of each type of signs in their relationship with the concurrent behavior models emerging from the analysis of the implementations of the Dining Philosophers on each CT tool as defined on Step 4.

Designer's metacommunication about CS concepts in CT tools has a interesting feature, since their message uses the set of (Computer Science) concepts being learned by the users to communicate them their message. Each of such tools have distinct characterizations of the implemented CS concepts, which are partially communicated in order to keep the activity fun and engaging [7]. Novice programmers are being presented to CS concepts in this context and thus must deal with partial

conceptualizations from which users may face issues when in need to implement problems outside the scope of the conceptualizations of the tool, and their solutions, although correct, may not be suited for the conceptualizations on the tool they are using, which may lead to users' frustration, thus impacting his/her learning.

In this context, we consider that designers should be aware of the effects of the signs they choose to communicate CS concepts which have been implemented in these tools. Thus, semiotic engineering helps us in two aspects. First, by identifying and discussing how these tools communicate their own implemented models of concurrent behavior. Second, considering that there is a great chance that students with different background may enroll in the same CS course during graduation, semiotic engineering helps us to provide instructors, directly, and students, indirectly, awareness on how distinct tools describe similar CS concepts, thus supporting both learners and instructors in capturing the essence of the concepts.

Step 6. After having a clear picture of the model of concurrent behavior from Step 4 and the designers' metacommunication message we proceeded to a contrast between the perceived model of concurrency and the metacommunication message of concurrency. This allows us to identify details about conceptualizations of concurrency, inconsistencies about the perceived concurrency and how users are communicated about it, among other details we describe in Section VI.

IV. ANALYZED TOOLS

For this research, we analyzed five CT tools: AgentSheets 4.0, Scratch 2.0, Greenfoot 2.4, NetLogo 5.1.0 and Alice 3.0. The tools themselves are built from distinct perspectives in mind and with distinct set of primitives to create programs. In general, all tools analyzed are centered on the programming of an element, a scenario or environment on which this element is placed, and a set of programming constructs which allow the user to manipulate it. As a reference, Table I shows the relationship between these elements, which are described later.

We applied the same framework to categorize CT learning environments as defined by Kelleher and Pausch [19] and analyzed these tools to evidence their similarities and differences. This characterization is presented in summary in Table II. We note that although very extensive, this framework could leave behind some characteristics that newer environments present for users to support programming. For instance, all tools we analyzed have some type of markup for programming constructs, being it colored syntax or even blocks that differentiate programming constructs. Despite of that, this framework provides to us an interesting analysis of each tool, which we discuss next.

AgentSheets is a CT tool whose programming is based on events, which the user defines for each "agent" in the "worksheet". An agent is a programmable object in which we can act upon. To this extent, the programming language constructs are designed to manipulate these agents by providing primitives such as "move" and "see", which create a connection with the physical world. This language is manipulated using images with a color affordance to represent conditions (what the agent perceives in the environment) and actions (how the agent acts on the environment) agents can perform. The tool only allows to place conditions and actions in their respective places in a rule, which is one command one agent evaluates for execution. Once programmed, we can manipulate an agent during the execution of the program, which is related to the liveness property of concurrent systems [19]. Lastly, it provides a tool for debugging, called Conversational Programming, which allows users to analyze the running code and make changes according to user's needs.

TABLE I. RELATIONSHIP BETWEEN ELEMENTS OF CT TOOLS ANALYZED

	Programmable element	**Scenario**	**Programming Constructs**
AgentSheets 4.0	Agent	Worksheet	Conditions/ Actions/ Triggers
Scratch 2	Sprite	Scene	Blocks
Greenfoot 2.4	Actor	World	Methods
NetLogo 5.1.0	Agent	Interface/ View	Functions/ Procedures
Alice 3	Class	Scene	Methods

Scratch is similar regarding the style of programming and it is based on the manipulation of objects (here called sprites). There are also commands which match physical behaviors of sprites, such as "move" and "turn". However, its programming language shares some resemblance with common programming languages, providing for and while loops, and conditionals which can be employed to manipulate sprites on the screen. These commands are grouped in categories for the type of behavior they allow to manipulate. These commands, which are called blocks, have a special shape which prevents the user from placing them in an incorrect position. This prevents syntax errors during programming. Scratch does not explicitly provide a debugging tool, although users can run scripts (which are pieces of code) individually, allowing them to look for errors locally.

Although projected to be an educational environment, Greenfoot goes in a different direction in the sense of the style of programming. Since it uses Java as the programming language, it provides an object-oriented approach, with all the power of this programing language, thus having all kinds of loops, conditional, parameters, procedures and user-defined data types. To build a program, users can type text for most of the coding process, although it is possible to place actors (which are graphical representations of classes) in a "world". This tool provides few ways to avoid syntax errors, by coloring parts of the code. However, most of the written code does not have this support. Apart from that, there is an API with methods that allow users to manipulate actors, such as "move" and "turn", like in Scratch.

TABLE II. COMPARISON BETWEEN ANALYZED PROGRAMMING LANGUAGES BASED ON FRAMEWORK FROM [19].

	AgentSheets 4.0	**Scratch 2**	**Greenfoot 2.4**	**NetLogo 5.0.1**	**Alice 3**
Style of Programming	event-based; object-based	Event-based; object-based	object-oriented	event-based; procedural	object-based
Programming constructs	procedures or methods; variables; parameters	for; while; count loop; variables; parameters; procedures or methods; conditional	for; while; count loop; variables; parameters; procedures or methods; conditional; user-defined data types	conditional; for; while; count loop; variables; parameters; procedures/ methods; user defined data types	conditional; for; while; count loop; variables; parameters; procedures/ methods
Representation of Code	pictures	pictures	text	text	pictures
Construction of Programs	assembling graphical objects;	assembling graphical objects;	typing code; assembling graphical objects	typing code	assembling graphical objects;
Support to Understand Programs	debugging; physical interpretation; liveness	physical interpretation; liveness	debugging support; physical interpretation	debugging support; physical interpretation; liveness	physical interpretation; liveness;
Preventing Syntax Errors	physical shape affordance; selection from valid options; dropping only in valid location	physical shape affordance; syntax directed editing; selection from valid options; dropping only in valid location	*none*	*none*	physical shape affordance; selection from valid options; dropping in valid location
Designing Accessible Languages	limiting the domain; user-centered keywords	Limiting the domain; remove unnecessary punctuation; user-centered keywords	limiting the domain; user-centered keywords	limiting the domain; user centered keywords; remove unnecessary punctuation	limiting the domain; removing unnecessary punctuation; user centered keywords;
Support Communication	network-shared	network-shared	*none*	*none*	*none*
Choice of task	fun and motivating; educational	fun and motivating; educational	fun and motivating; educational	fun and motivating; educational	fun and motivating; educational

NetLogo is a tool which employs a style of programming more distant from the other tools we analyzed. It is also event-based, but it employs a procedural approach to programming. Users must define functions and define on it each aspect of the game. Although the programming language has functions that provide connection to the graphical aspect of the program such as `forward` or `fd`, which move the agents forward, these are only visible once the user plays the game, which means there is no way to manipulate graphical objects other than by textually coding them. However, NetLogo allows us to change the execution at runtime, by employing variables that can be set during the run of the program.

Alice is an object-based tool. Despite the programming language being a simplification of Java, it allows the user to manipulate only methods from actors it brings. However, new methods can be created, and many Alice-exclusive primitives. These provide conditionals, loops, variables, parameters and procedures which allow the user to manipulate actors in the environment (here called scene). The construction of programs is based on the graphical manipulation of both methods and primitives of the programming language and actors from a set of actors provided by Alice. These are mostly characters and abstract elements such as rectangles and spheres, which limit the domain of the programming activity to storytelling purposes.

V. ANALYSIS OF IMPLEMENTATIONS IN CT TOOLS

As defined in Step 3 in Section III, we implemented in each CT tool multiple versions of the Dining Philosophers obtained from Step 2. This allowed us to analyze if different configurations of the code in each CT tool would result in different concurrent behaviors. Each configuration used a new subset of programming constructs from these CT tools to describe each step of the Dining Philosophers. Table III shows a summary of the respective number of configurations and the programming constructs explored to create them. It is also interesting to note how each programming construct was employed in each CT tool for each version.

On AgentSheets we created 8 versions of the game. The versions differ mostly regarding the representation of agents and the representation of the fork-taking process. Regarding the representation of agents, AgentSheets provides us with two ways of defining them: by one agent with multiple depictions or by creating multiple agents, one for each fork and philosopher (Fig. 1). This is possible because there are programming primitives which allow us to operate on instances of agents or depictions. As we could note, exchanging these primitives did not change the result of the program. Also, we varied how the fork was taken. It is possible to either define one rule on each philosopher for the “taking” process, in case we use one agent to represent each philosopher, or a set of rules

on one agent, if we use depictions. Also, we could apply rules to the fork to guarantee that it will be eventually taken, and to assure it is not taken multiple times. This also did not change the result of the game when compared with other versions created previously in the CT tool during our analysis. Thus, we concluded that in AgentSheets the primitives have no distinct

TABLE III. VERSIONS AND VARIATIONS APPLIED TO PROGRAMS CREATED IN EACH CT TOOL.

	# of versions	Variations
AgentSheets 4.0	8	single/multiple agents to represent forks/philosophers; single/multiple rules for fork-taking process; reordering of agents on environment
Scratch 2.0	5	using broadcast or variables to change the state of forks; changing position of conditional parameters and instructions; using clones
Greenfoot 2.4	5	single/ multiple actors to represent forks/ philosophers; changing method for philosopher to interact with fork; reordering placement of actors
NetLogo 5.1.0	5	using functions for each step of Dining Philosophers; not using functions for fork-taking process; creating each philosopher individually; using 'ask-concurrent' primitive
Alice 3	5	using `do in order/do together`; changing order of agents; changing order of fork-taking conditionals; using variables for fork state;

influence in the result of the game. This difference is implicit in the way agents are positioned in the game environment. We then created a version in which agents are positioned in different places from the other versions, and this changed the game outcome, since the philosophers who 'took' the fork were different from the other versions of the same game.

On Scratch, sprites are the main element to program, which resemble the concept of agent on AgentSheets. Additionally, it is possible to program in the stage (the worksheet, on AgentSheets. However, commands acting on sprites, which were needed to implement the problem we defined, must be placed on the sprite's script area. Thus, it was not possible to describe any of the needed behaviors only using the stage area. Additionally, to make a sprite to respond to an action from other sprite, Scratch provides the "broadcast" command. We applied this command in our versions to allow forks and philosophers to respond and change states. Given the procedural paradigm in this programming language, we changed positions of instructions and conditionals to check if there was any difference regarding the result of the game. Also, we employed the use of variables to define the state of the fork, to analyze the game behavior (Fig. 2). Only in this case the result differed from the other cases. In a further investigation, we discovered that the broadcast command is treated in a different manner than other primitives and since we did not need to use it to explicitly change the state of the fork, we got a different result. Additionally, we used the concept of clones Scratch provides to create sprites at runtime, with no difference in the outcome of the program when compared to the use of "broadcast" command. We created a total of 5 versions in this CT tool.

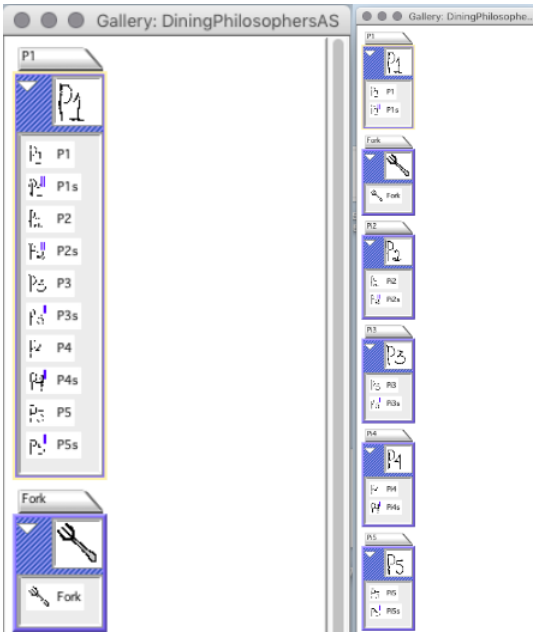

Fig. 1. Example of distinct representations of a Philosopher on AgentSheets; in the left, there is one agent with depictions for instances of philosophers and their state; in the right, each philosopher has its own agent on AgentSheets.

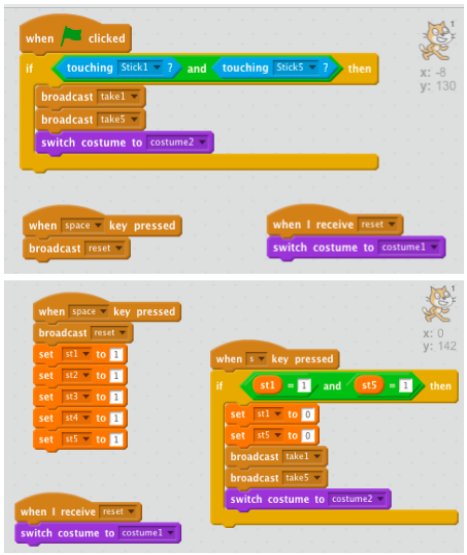

Fig. 2 Two representations of the fork taking: Above, using broadcast only and the second; below, using variables to represent the states of the fork.

On Greenfoot, we created 5 versions of the game. These versions varied according to how we defined the actors, if using one philosopher class (actor) with different images or multiple classes (actors); adding a restriction to fork's behavior to guarantee that the fork is taken and it is taken by only one actor; using a different method available to check if the fork was taken than it was used in the initial version; and changing the order of definition of actors and their placement in the environment (world). The first 4 versions we mentioned did not affect the result. It was only by changing the actors positioning order that we could see a change in the result of the game (Fig. 3). This allowed us to infer, later, the concurrent behavior of the Dining Philosophers on Greenfoot.

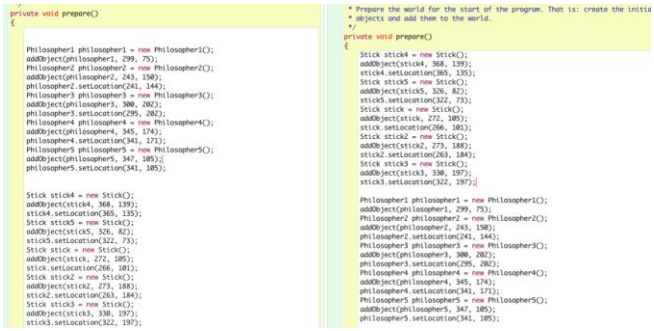

Fig. 3 Different sequence of actors positioning on Greenfoot

On NetLogo, we created 5 versions of the game: one using functions at all steps of the definition of the Dining Philosophers; one not using functions on the fork taking process; one using a one-by-one definition and placement of agents; one using a button and an implicit call for agents to pick forks (Fig. 4); and one using the "ask-concurrent" primitive. From these variations, the only change was perceived when using 'ask-concurrent' on one of the methods.

Finally, on Alice, we created 5 versions, varying the use of "`do in order`" and "`do together`" for programming the behavior of philosophers (Fig. 5); varying the use of `do in order` inside or outside if's which describe the fork-taking process; and using variables to discriminate the state of the forks during the fork-taking process. In our analysis, we could see that do together and `do in order` change the output of the program, specially when changing it to control the execution of each "if" which controls each philosopher process of taking the forks.

Creating these versions allowed us to have a clearer picture of the influence of primitives used to create instances of the Dining Philosophers in these tools and hints about the influence of the CT tools interface and interaction signs, in the sense of a semiotic inspection we performed on our proposed methodology.

VI. FINDINGS

In this section, we present the results of the application of the method described in the previous section. In this section, we present the rationale about the model of concurrent behavior.

```
;; grab the left fork. set its owner to me and move it
to acquire-left  ;; philosopher procedure
  ask left-fork [
    set owner myself ;owner is a left-fork variable. "myse
    move-to owner
    set heading [heading] of owner
    rt 8  fd 2  ; right turn + forward
  ]
end

;; grab the right fork. set its owner to me and move it
to acquire-right  ;; philosopher procedure
  ask right-fork [
    set owner myself ;owner is a left-fork variable. "myse
    move-to owner
    set heading [heading] of owner
    lt 8  fd 2 ; left turn + forward (to be more visible)
  ]
end
```

```
to go
  ask  philosophers [
    show count philosophers

    if [owner] of left-fork = nobody and [owner] of right-fork = nobody
    [
      ask left-fork [
        set owner myself ;owner is a left-fork variable. "myself" means
        move-to owner ; move to the owner
        set heading [heading] of owner
        lt 8  fd 2  ; right turn + forward
      ]


      ask right-fork [
        set owner myself ;owner is a right-fork variable. "myself" means
        move-to owner
        set heading [heading] of owner
        rt 8  fd 2 ; left turn + forward (to be more visible)
      ]
    ]
  ]
  ;recolor
  tick
end
```

Fig. 4 Use of functions to represent the fork taking above; and less functions below, on NetLogo.

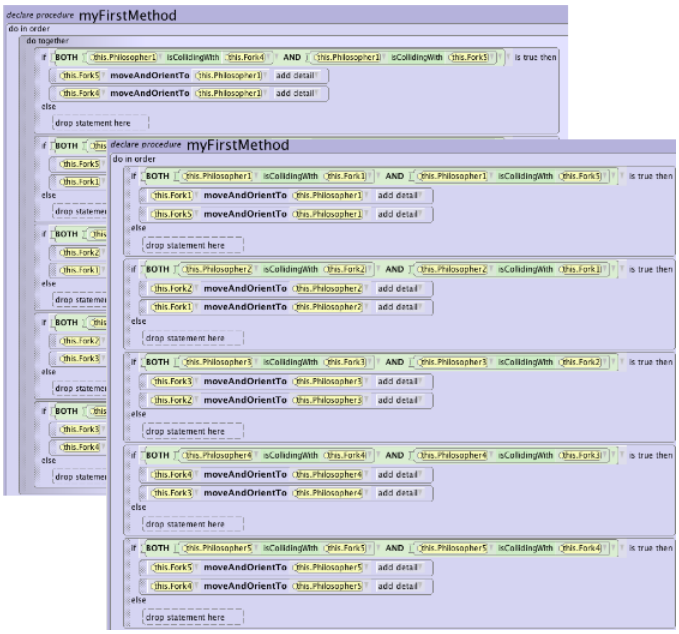

Fig. 5 Alice and the use of `do in order` and do together.

Following we present the metacommunication message of each tool as revealed by SIM, and finally, the contrasts regarding the concurrent behavior modeled and the result of SIM.

A. *Models of Concurrent Behavior*

The models of concurrent behavior we describe here result from the analysis of outputs generated by the implemented versions of the Dining Philosophers. As previously mentioned, each CT tool allows us to create multiple versions of the same game, due to the possible ways of representing each part of the pseudo-code in each tool. However, overall these variations did not influence the final model of concurrent behaviors we inferred from the outputs, although they allowed us to perceive the existence of distinct models in the same CT tool.

For all tools, influences on the program outcome were noticed when placing agents on the simulation environment and when programming behaviors for an agent. The placement of agents is different in each tool we analyzed. The spreadsheet-based environment of AgentSheets, for instance, does not allow agents to be placed in a perfect circle. However, it is possible to place agents in the same relative positions as in other environments, which allows us to define the similar behavior of each philosopher and forks equally, in the sense of the pseudo-code. Figure 6 shows this situation between AgentSheets and Scratch.

Regarding the definition of agents' behavior, the paradigms each CT tool employs allow the user to define the behavior of agents in many ways. On AgentSheets, Scratch and GreenFoot, behaviors are programmed inside agents, with code being defined *in* and *for* an agent/sprite/actor. On NetLogo and Alice, behavior is defined outside of agents, inside the *Code* tab, for NetLogo, and inside the *Scene*, for Alice. This last characteristic allows users to have more control over how agents are visited, thus influencing the concurrent behavior of the game or simulation. Figure 7 presents a comparison between Greenfoot and Alice coding strategy, in which on the former, we have the "Philosopher" class containing the behavior of the Philosopher, through the "act" method, while on the latter, behaviors are described on the "Scene".

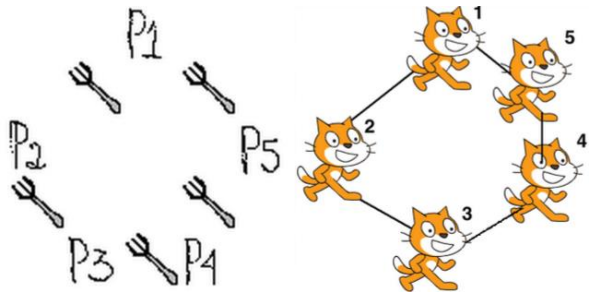

Fig. 6. Simulation environment for AgentSheets on the left, and Scratch, on the right.

In order to discuss these distinct models in a consistent manner, we use Petri Nets. Figure 8 shows the Petri Net for our version of the Dining Philosophers. A Petri Net consists of places and transitions. Places can be states or parts of the system, depending on which aspect of the system we want to model. Transitions are triggers to the occurrence of these events. This is described in a "run" of the Petri Net, in which we describe how each of these transitions occur. These transitions can occur simultaneously, given they have the correct premises to run.

In our model of the Dining Philosophers, forks are represented as places labeled "F_x", were "x" ranges from one to five, and containing one token each, to represent its availability. Philosophers are also places named "P_x", also with "x" ranging from one to five. Transitions are rectangles unlabeled and represent the act of getting the fork for each philosopher. In this version, we modeled only the process of getting the fork which is the concurrent behavior we wanted to analyze in each CT tool. Thus, there is no repetition in the fork-taking attempt which the original Dining Philosophers problem states. We employ this model as a basis for discussing the models of concurrent behavior we unveiled from our analysis.

After the analysis of the running of the implementations on the CT tools three models of concurrent behavior emerged. These models are mainly based on a deterministic running of the processes, although there are variations among them. The first, presented on Figure 9, is a sequential solution, which resembles a mutual exclusion solution, and AgentSheets and Scratch implement it. In this case, each process "acquires" a lock and must wait until the "lock" is released.

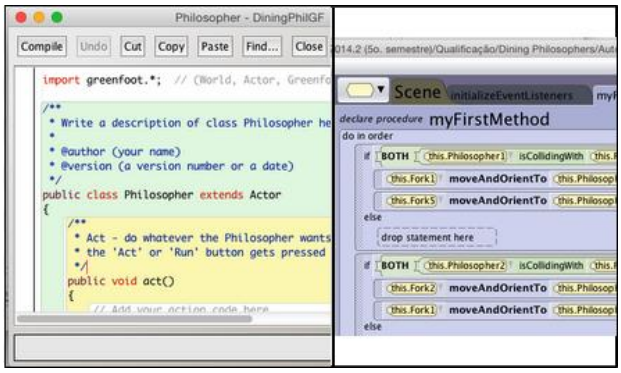

Fig. 7. Greenfoot code on Philosopher on the left, and Alice's code about the Philosopher, on Scene, on the right.

On AgentSheets and Scratch, the interpreter controls this locking mechanism, which allows one agent/sprite to run at a single time (a solution for a critical section problem, common in concurrent programming [2]). However, there are differences on which agent is locked first. On AgentSheets, the locked agent is the last agent inserted on the worksheet and using one agent with distinct depictions (the "costume"-equivalent of Scratch on AgentSheets) or using distinct agents did not change this order. On Scratch, it is the last manipulated agent that runs first. In both environments, this characteristic is external to the implementation, thus the "Fixed" behavior we describe in this model. The model is represented on Figure 9. On it, we see that transitions are run sequentially, given the model predicts a sequenciality in the fork acquisition for each agent/sprite in these tools.

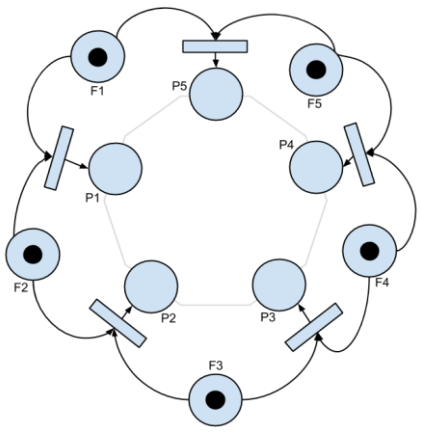

Fig. 8. Dining Philosopher modeled using Petri Net modeling notation.

Another interesting thing to notice in this case is the use of the `broadcast` command. One of the implementations uses broadcast to make the "Fork" sprite to hide, in order to represent the fork being taken. However, on Scratch, using this command does not "lock" other sprites. This generates a contrast between the expected output and the occurred one, since all the other sprites will assume there are forks available for them, when in fact this is not the case since other sprites have already initiated the "fork taking" process.

A second model can be perceived on NetLogo and Greenfoot, and it is presented on Figure 10. On these tools, there is also a sequential behavior, but there is a previous step regarding the choice of which agent always runs first. On Greenfoot, the order of execution is defined by the order the agents are created and positioned which is defined by how they are written on the `World` code. However, this step is not explicit in the programming activity, but it is possible to see it since the programmer is able to access the `World`'s code and change this order. On NetLogo, when using `ask`, this is not possible, since the agents execute on random order, even though they are created in a particular order as one of our implementations showed. This is in line with Ben Ari [3], who claims that this is a behavior of concurrent programs, thus NetLogo is correctly representing the expected concurrent behavior. Other tools did not present the same outcome. However, these are part of the external factors that change how the concurrency issue is solved but are not part of the problem.

Alice implements the third model, whose interesting characteristic is the possibility of exploring the sequential, or parallel behavior of the execution. In order to allow this, this tool provides two commands, `do in order` and `do it together`, which changes the way the commands inside these instructions are interpreted[6]. The first one allows us to execute each command inside it sequentially, like in the model on Figure 11. The second one allows us to run the commands at the same time, which allows us to make sequence of commands run parallel, represented on Figure 11. This kind of control is not present in the other tools.

Nevertheless, two comments about these commands can be made regarding the results they allow us to express. First, depending on where we place these commands, we can get distinct behaviors. If we place all the behaviors (represented by if's) inside a `do in order`, it behaves as AgentSheets and Scratch in Figure 9, although each fork is taken individually, instead of them being taken at the same time on the simulation. If we use `do together` instead, we get a strange behavior, since all forks are taken simultaneously, as if they are all available.

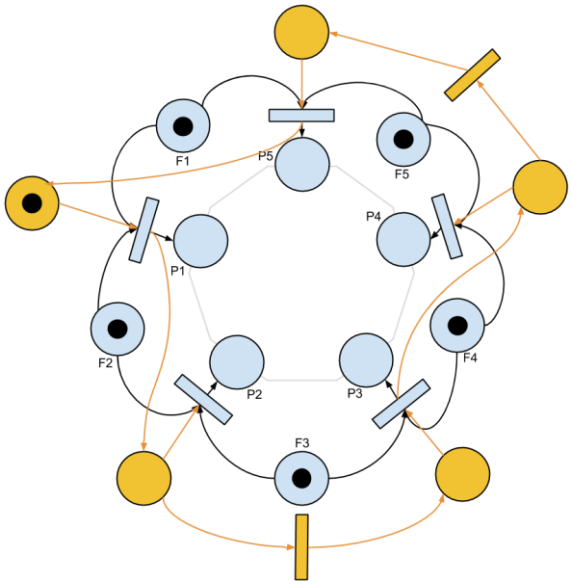

Fig. 9. Sequential model that emerges from AgentSheets and Scratch implementation analysis. Yellow elements represent the additions for tools to run the model

Additionally, the forks are taken by philosophers randomly, an effect similar to the NetLogo model. As we can see in Figure 10, this behavior changes the expected model, since we consider that each fork is available even when it was taken before. This is not correct for the model and it is not a correct solution of the problem. This happens because in this situation, each condition is checked at the same time, which does not give time for the simulation state to change for each philosopher. This does not happen if we place only the command `moveAndOrientTo`(), which represents the taking of the fork, inside the `do together` on each `if` rule. In this case, both forks are taken at the same time, which is the same way AgentSheets and Scratch perform.

B. Metacommunication Message of CT Tools

The models of concurrent behavior depicted earlier provide us a clearer picture of the underlying concurrency concepts, which are presented to the user via designers' messages. In this subsection, we contrast these models with findings from the application of SIM on CT tools. We categorize our findings regarding the strategies for communicating concurrency that tools are implicitly employing, thus visualizing the conceptualizations of concurrency that occurs in these CT tools.

[6] http://academy.oracle.com/self-study/alice/alice_7_3.html

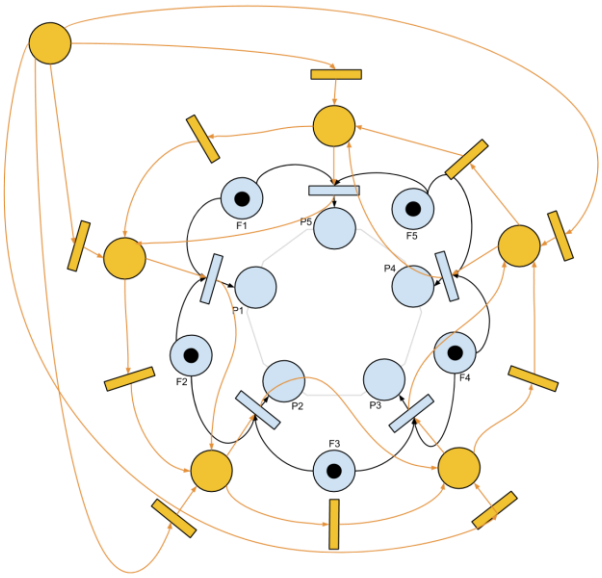

Fig. 10. Model of concurrent behavior for Greenfoot and NetLogo

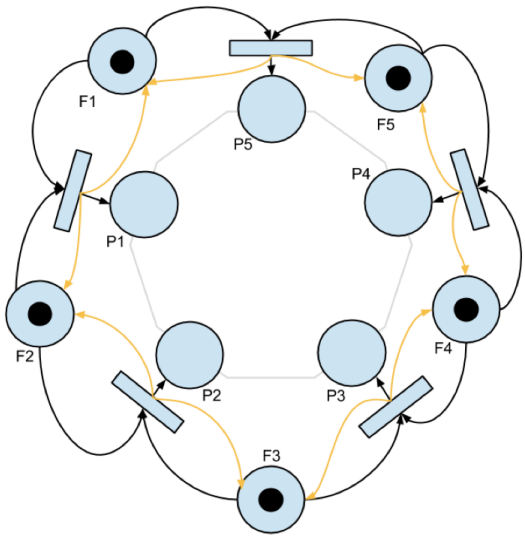

Fig. 11. Alice`s model when using "Do Together" outside if`s.

1) Acting/Triggering of processes

In a concurrent environment, a trigger is expected to start the execution of the model. From a "natural world" perspective, these triggers can happen due to causal events, as can be seen on the Petri Net on Figure 3, for the Dining Philosophers problem. On the "computational world" which these tools implement triggers are mostly bound to causal events, as would be natural. Despite that, three variations on how they communicate this to the user can be found on these tools, which are:

• **Implicit**: there is a unique trigger that cannot be changed other than building upon it. Greenfoot has an example of this kind. Each object on the simulation has a "act" method, which according to its documentation "... is called by the Greenfoot framework to give actors a chance to perform some action. At each action step in the environment, each object's act method is invoked, in unspecified order."[7] This is reinforced by the interface, when we right-click an agent and see the `act()` as one of the options. Clicking on it starts the running of the single agent clicked before. However, there are no explicit references to this behavior in static and dynamic signs; it is only on metalinguistic signs that we are able to understand what `act()` means clearly. The downside of this option is that it would drive the user to create unnecessary abstraction to deal with specific types of triggers, which could lead to errors when these created triggers overlap with the model provided by the tool.

• **Default**: AgentSheets, Scratch and Alice shows one default trigger for starting the running sequence, in distinct ways. AgentSheets and Scratch own a set of triggers (or events, on Scratch) that can be used to start some execution. Scratch has no default trigger depicted on the coding area. However, the "flag" on top of the Stage area blinks when the program starts to run indicating that the flag can be used as trigger. Alice's interface has a default method for running the Scene, which is the "Scene Activation Listener". Despite of this any type of code can be added, including a specific trigger for any moment in time, for instance. The same issue about creating unnecessary abstractions that could overlap with the default trigger that happens on implicit models could happen to default models, if they are not communicated explicitly as options of triggering.

• **Open**: Open triggers are the most interesting since they allow the user to explore different possibilities of starting the execution of agents, which can minimize non-desired concurrent behaviors. However, they do not allow showing trigger options that could happen in a real environment. On NetLogo, there is no evident trigger that could be used, and interface signs provide no clue about what could be used to start the simulation. However, according to the documentation, "ask" is the way we can run commands on processes, which means this is NetLogo "default" way of running an agent behavior, but not an entire simulation.

2) Ordering of processes

The models analyzed reveal that the common concurrency solution is based on a turn-taking mechanism, similar to a locking mechanism, common in concurrency solutions [3]. Each model provides a specific way of dealing with this mechanism. Our semiotic analysis shows that the communication of this mechanism does not reflect the proposed ordering by these tools nor provides a way to reviewing this ordering. Additionally, the order presented by the user, that of the scenes/stages and the gallery of agents deviates the user from a semiosis that leads to the correct interpretation of the ordering that the tool implements. Only the multiple execution of the program allows the user to correctly describe how each agent is visited. Additionally, some of the tools allow the user to run only parts of the code or one agent's behavior, which allows the user to see for him how one agent would behave if only him runs.

For the tools with the "Sequential Fixed" model of concurrent behavior, there is a distinction on how to communicate ordering for each tool. On AgentSheets, interface and worksheet provide us with two distinct orders of agents. None of them are used on the model, since the actual order comes from the stack created from the last to the first agent inserted on the worksheet, and not the order of philosophers inserted on the worksheet as described by the user. Metalinguistic signs on the documentation of AS describe the running of simulation when the Run button is pressed: "The

[7] https://www.greenfoot.org/files/javadoc/#act

Run button starts the simulation […]. When the simulation is running, all agents with a While Running method execute the behavior defined in that method." It does not describe how agents are visited. Scratch also does not communicate a specific sequence of execution. However, it is possible to run a single agent's code to see how it behaves. However, on one of the implemented versions, in which we used "broadcast" as a mechanism for the action of getting the forks, using this would make the user mistakenly think that the implementation works correctly, when in fact, it does not, since when running the simulation, all agents mistakenly get forks, due to a specific behavior of the broadcast script that leads to this error.

On tools with the "Sequential Flexible" model, NetLogo and Greenfoot, there is also a difference on how these tools deal with ordering. Greenfoot has similar signs of execution when looking at static signs as AgentSheets. Both have a Run, Step, Stop and a slider for controlling the execution NetLogo also has. However, neither of them provides signs to the order of processes executed. On Greenfoot, the `World` class contains the code which reveals the ordering of agents' execution during the simulation. When adding an agent via interface, the user does not see this code and therefore cannot reach a correct conclusion about the model implemented. On NetLogo the visiting of agents is random and no signs, metalinguistic or static, allow the user to capture this behavior. Only by running the simulation multiple times is that the user can see such behavior.

The last model, implemented by Alice, is an open model since it allows the user to choose which model to run. However, as noted before, this choice must be made carefully in order not to create live lock situations as seen in one of the game's implementations on Alice, when in each "step" of the game, the same fork "x" is taken by distinct philosophers. Regarding the communication of these issues, metalinguistic signs present on Alice`s tutorial on Oracle, accessible by the link on help menu provide few clues about the range of behaviors regarding the ordering of processes visiting. However, this is minimized given the way code is written, in the Scene methods that control execution. The user explicitly states the order of execution. When using the `do together`, user will perceive misplacing when running the project. However, as in some of the other tools, user cannot run one process at a time to analyze the behavior of agents, which makes it difficult for users to visualize how a specific process behaves in comparison to the sum of the activities. However, this issue is diminished in the sequential model, when using `do in order`, since users can see each step being performed as an animation.

3) Atomicity of a cycle

Another characteristic of concurrency is the concept of a process atomicity. In each model presented, we describe the philosophers and forks as places on the tool. Although the model runs concurrently, the processes are independent and equipotent units of the entire program. On the tools analyzed, although there the concept of process is present, under the name of "agents", "sprites" or "actors", there is no consistency among static, dynamic and metalinguistic signs regarding the atomicity of a process.

On AgentSheets, Greenfoot and Scratch the concept of cycle or turn is bound to a complete run over all agents. However, the slider in these tools (which allows us to see this characteristic) is misleading in the sense that it allows the user to consider that it would slow down or speed up the running of a single agent at a given time. On Alice this happens differently, since the slider allows the user to change the frame speed, or how fast agents perform their behaviors. In the "`do in order`" version of Alice implementations of the problem, it is possible to visualize each agent performing one step at a time whilst using "do together" inside the if's it is not possible.

On NetLogo, metalinguistic and static signs on the interface show the `tick` function, which represents a step of the execution on NetLogo and when called, it adds 1 to a counter. However, a `tick` can only be called on observer procedures. An observer is an agent external to the simulation. This characteristic would guide a novice programmer to infer that this is the only possible granularity for a process execution, which is not true outside of NetLogo.

4) Concurrent programming ontology

The categories previously described provide us with a picture of how concurrency emerge from these tools. Additionally, the programming languages on these tools allow us to create and manipulate concurrent concepts. From our analysis, two subcategories regarding programming constructs emerged. The first one talks about the representations of process and concurrent tasks on these tools, while the second calls the attention for the visual component that influences how concurrent programming is implemented.

Programming constructs: Expressing concurrent behavior in these tools vary, at the programming language perspective, since each tool employs a specific programming paradigm, but tools also share specific characteristics regarding the characterization of a process. The first noticeable element is the naming of a process. AgentSheets and NetLogo name it an agent. Scratch name it Sprite; Greenfoot, actor; and Alice, Class, which is closer to the object oriented paradigm, or the Actor model of concurrent programming [8]. The way a programmer manipulates these elements also varies. On AgentSheets and Greenfoot, it is possible to create instances of the process, which contains all the behaviors programmed earlier. On Scratch, like in the other tools, it is possible to create a copy of a process, although this copy has no connection to its sibling. It is indeed possible to clone a process on runtime, but this agent has no identification inside the tool, which demands the user to create an additional abstraction to name each clone of an agent and to manipulate its behavior separately, if needed. In the case of the Dining Philosophers, to use clones, we would still need to differentiate each agent to identify which forks are next to each agent.

Concerning behaviors, there are also distinctions among the tools. On AgentSheets, Scratch and Greenfoot behaviors are programmed in the processes. Scratch and Greenfoot can have behaviors programmed on the scenario. On NetLogo and Alice, the simulation behavior is not part of the agent. Considering concurrency, allowing us to control the process from outside allows the programmer to have control how processes run, since we can program its behavior. On NetLogo, this is possible if the user calls `ask` for each agent created; on Alice, the tool provides the "`do in order`" and "`do together`" constructs. On Scratch, Greenfoot and AgentSheets, the programmer would need to explicitly order the execution on top of the present order, which is creating an unnecessary abstraction in this case.

Visual component: The visual component in these tools allows the user to see its programming coming to life and it is an important aspect of these tools for the promotion of programming. Concerning concurrent programming our analysis shows an intersection between the visual component and the programming activity, which by one hand makes the concurrent behavior easily visible, but on the other hand creates an extra layer of abstraction for the user to program concurrency, since the programmer needs to consider that the result will be visually perceptible, considering that the visual element is part of these tools.

On the tools analyzed, programming languages are bound to the visual component at some part of the programming activity. On AgentSheets, for instance, to "see" a fork, we need to place it next to two philosophers, since the `See` construct demands that agents need to be in one of the adjacent cells to be visible. On Scratch, Greenfoot and Alice, this demands that the users need to be touching for us to use the programming constructs provided by the tools. If philosophers and forks are not touching each other, he/she needs to create additional abstractions to achieve the behavior of getting the fork, which can be challenging for a novice programmer. On NetLogo, this is the first solution available for the programmer and in this case, there is no changing of the paradigm for programming this behavior.

VII. DISCUSSIONS

This paper investigated conceptualizations of concurrency on five CT tools. The systematic evaluation we present allows us to discuss, in the perspective of computational thinking acquisition, the role of tools, which along with the curricula and teachers, also communicate CS concepts to novice programmers. As our research shows, there are many aspects of concurrency concepts that emerge from interface signs designers expose to users, whether intentionally or not. These concepts, however, are usually bound to one kind of concurrent behavior. As mentioned in Section VI, only Alice would make explicit different concurrent behaviors using different programming constructs. Furthermore, as we also discussed, these concurrent behaviors are in general inconsistent with concurrency as explored in CS, since few consider, for instance, the nondeterminism during the execution of the program. In this sense NetLogo brings the closest approach, in line with these concepts of nondeterminism which are important to talk about concurrency [2].

Our analysis of the contrast between models of concurrent behavior and SIM allowed us to evidence the conceptualizations of concurrency each tool provides to users. The categories of concurrent communication strategies points that these conceptualizations vary among tools, and concurrency itself is underexplored by them in a higher level of representation. This can be explained since exploring concurrency is not an explicit goal of these tools. However, they bring concurrent conceptualizations demanded to run programs. Our argument is that these needs to be better explored to avoid all sort of difficulties which could be brought through the manipulation of these concepts by learners.

A. Mappings findings with literature

From the literature, we can map our findings regarding concurrency learning. First, regarding triggers, the understanding of the triggering mechanisms can be related to the existence of a clock in the execution, which controls when processes will start running. This is closely related to the understanding about how computers deal with concurrent processes. Difficulties in understanding this particular point are reported in the literature, for instance when students assume there is a global clock where none exists [9]. For these types of students, default or implicit triggers could reinforce this wrong assumption. Also, this relates with the concept of simultaneous execution, when, taking AgentSheets and Scratch for instance, these triggers are spread over different processes. This could also cause problems for students, like the issues of Type II concurrency, when talking about concurrency on the agent itself.

Many papers discuss students' difficulties regarding ordering of processes, calling the attention for difficulties on dealing with coordination [10], or the program execution itself [11]. As research shows, students may not understand correctly the computational model (or world), which may create difficulties in assessing the correctness of the program. On a study about high school students' knowledge structure and dynamic of the construction process of concurrent programs [12], findings show that dynamics of the execution of programs is a place for inadequate assumptions, especially when these behaviors are not similar to previous knowledge gained by the novice programmers, and students engaged in these activities not using pieces of knowledge related to concurrency. Despite using a specific tool to study concurrency, this work reveals that students may find trouble when dealing with the execution of concurrent programs. As we found in our research, signs do not expose details about the models of concurrent behavior, which could increase the type of misconceptions found in this and other research mentioned earlier in this section. For students, the mental simulation of the computational model can be difficult due to many possible scenarios in each run [13]. The misconceptions of the computational models are also allied with other discussions regarding the understanding of the concept of model itself [10]. Also the same issues are present when analyzing the use of a tool specifically designed for dealing with concurrent behavior when Resnick analyzes the use of MultiLogo by students [14].

The atomicity of the cycle is also an issue among students when learning about concurrency. This could cause problem decomposition issues [14], since students do not understand what are the limits of an execution, as the tools analyzed do not communicate it clearly, by communicating a step, or atomic operation, as the sequence of all processes runs, which could be related to the causal behavior of the Petri Net, but does not help in contrasting it with the computational world, whose contrast is the most valuable lesson. Specially for students that deal with sequential processes, these intermediary outputs would help them in understanding the concurrent execution [10].

Lastly, the programming ontology also shows that constructs and the visual component present in these tools can influence both the perception of concurrent execution and how users program concurrent behavior. Regarding the programming constructs, the role of the representation of concurrency has already been discussed. Different models, in this sense, may guide the user to specific characteristics of the concurrency domain [8]. However, the agent based programming has been considered an interesting way to deal with microworlds and their concurrent behavior [13]. Regarding the visual component that influences the CT tools analyzed, they can be also related to the representation chosen

for representing processes, to maintain its closeness to the real world. However, as noted in our implementations, their restrictions over the real world may impact users' expression about concurrent behavior. This could be seen when using "touching" features of these tools, which in some of the tools is only possible if images are overlapping on the screen. The programming ontology chosen is not without reason. Designers of these tools have specific goals in mind when these were developed, to fulfill their purposes.

B. Limitations of the study and future directions for research

Although this research allows us to reveal conceptualizations of concurrency emerging from these CT tools, limitations regarding its reach must be considered. First, the case chosen allowed us to explore a concurrency situation where resources are being shared (in this case, forks). Other aspects of concurrency could be explored with other cases, such as mechanisms or algorithms for dealing with concurrent issues. However, the quality of a qualitative research is in the in-depth exploration of the data and its meaning. The same can be said for the tools chosen. Additionally, other tools can reveal additional conceptualizations of concurrency, and complement the results shown here. Another aspect is regarded to the execution of SIM after implementations were created. This previous implementation in the CT tools served as preparation for establishing the focus of analysis of the method, as it is recommended [2].

However, it is important to note that our semiotic perspective reveals one side of the picture on how students may be influenced by how CS concepts are presented to them. Other perspectives can be taken for analysis. Cognitive aspects of these tools, for instance, could also be investigated. Considering that these tools provide a notation for expressing software, Cognitive Dimensions of Notations (CDN) framework could provide us interesting insights both about the tools and their notations. SIM were already combined with CDN framework before [15]. We could extend this method and focus on notations about CS concepts, to complement the results with our semiotic inspection. As a characteristic of the method employed for semiotic inspection, it only reveals the emission of the designers' metacommunication message. Further research can be conducted to investigate the reception of this message by users of these tools, by using CEM [2].

This paper adds to the discussions of the role of CT tools in the context of computational thinking acquisition and presents opportunities for exploration of concurrency concepts during the computational thinking acquisition process. The mapping with literature about students understanding of concurrency concepts, evidences that already explored misconceptions which pre-university students may have can be impacted by the tools they may use, either previously to their encounter with these tools or not. Considering the widespread use of CT tools in the Computational Thinking acquisition activity, especially on pre-college education, it is crucial to understand the role of tools and what biases they might be creating on to-be professionals.

VIII. ACKNOWLEDGEMENTS

Authors would like to thank CAPES for its financial support, and Clarisse Sieckenius de Souza for providing valuable feedback to this research.

REFERENCES

[1]. M. Ben-Ari e Y. B. Kolikant. 1999. Thinking Parallel: The Process of Learning Concurrency. *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, ACM, 13–16. http://doi.org/10.1145/305786.305831

[2]. M. Ben-Ari. 2006. *Principles of Concurrent and Distributed Computing*. Pearson.

[3]. Y. B. Kolikant. 2004. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies* 60, 2, 243–268. http://doi.org/10.1016/j.ijhcs.2003.10.005

[4]. J. Esparza. 2010. A false history of true concurrency: From Petri to tools. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 180–186. http://doi.org/10.1007/978-3-642-16164-3_13

[5]. R. Feldhausen, S. Bell, e D. Andresen. 2014. Minimum Time , Maximum Effect : Introducing Parallel Computing in CS0 and STEM Outreach Activities Using Scratch. *XSEDE 2014*, 7.

[6]. J.J. Ferreira, C.S. de Souza, e L.C. de Castro Salgado. 2012. Combining cognitive, semiotic and discourse analysis to explore the power of notations in visual programming. *Proceedings of VL/HCC'2012*, 8.

[7]. E. Kraemer. 2010. Characterizing Comprehension of Concurrency Concepts. *PPIG 2010 - 22nd Annual Workshop*.

[8]. Z. Li e E. Kraemer. 2013. Programming with Concurrency: Threads, Actors, and Coroutines. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 1304–1311. http://doi.org/10.1109/IPDPSW.2013.193

[9]. J. Lönnberg, L. Malmi, e A. Berglund. 2008. Helping Students Debug Concurrent Programs. *Proceedings of the 8th International Conference on Computing Education Research*, ACM, 76–79. http://doi.org/10.1145/1595356.1595369

[10]. O. Meerbaum-Salant, M. Armoni, e M. Ben-Ari. 2010. Learning computer science concepts with scratch. *Proceedings of the Sixth international workshop on Computing education research - ICER '10*, 69. http://doi.org/10.1145/1839594.1839607

[11]. S. Papert. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books.

[12]. W. Reisig. 2013. *Understanding Petri Nets*. Springer.

[13]. A. Repenning e A. Ioannidou. 2008. Broadening participation through scalable game design. *ACM SIGCSE 2008*, 305. http://doi.org/10.1145/1352322.1352242

[14]. A. Repenning, D. Webb, e A. Ioannidou. 2010. Scalable game design and the development of a checklist for getting computational thinking into public schools. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ACM Press, 265. http://doi.org/10.1145/1734263.1734357

[15]. M. Resnick. 1991. MultiLogo: A Study of Children and COncurrent Programming. *Interactive Learning Environments* 1, 3, 153–170.

[16]. S. Schwarz e M. Ben-Ari. 2006. Why Don ' t They Do What We Want Them to Do ? September, 266–274.

[17]. C. S. de Souza, A. C. B. Garcia, C. Slaviero et al. 2011. Semiotic traces of computational thinking acquisition. *IS-EUD 2011*, Springer, 1–16. http://doi.org/10.1007/978-3-642-21530-8_13

[18]. C. S. de Souza e C. F. Leitão. 2009. *Semiotic Engineering Methods for Scientific Research in HCI*. Morgan & Claypool. http://doi.org/10.2200/S00173ED1V01Y200901HCI002

[19]. C. Kelleher and R. Pausch. 2005. Lowering the barriers to programming: A taxonomy of progarmming environments and languages for novice programmers. ACM Comput. Surv.. 37, 2. , 83-137.

[20]. M. Resnick 1994. Turtles, Termites and Traffic James: Explorations in Massively Parallel Microworlds. MIT Press, Cambridge MA, USA.

[21]. C.A.R. Hoare. 2004. Communicating Sequential Processes. Prentice Hall International.

[22]. C. Hewitt; P. Bishop; R. Steiger 1973. A Universal Modular Actor Formalism for Artificial Intelligence. Proceedings of 3rd international joint conference on Artificial intelligence, pp 235-245.