Experiences Implementing and Utilizing a Notional Machine in the Classroom

Paul E. Dickson
pauldick@buffalo.edu
University at Buffalo
Buffalo, NY, USA

Tim Richards
richards@cs.umass.edu
University of Massachusetts Amherst
Amherst, MA, USA

Brett A. Becker
brett.becker@ucd.ie
University College Dublin
Dublin, Ireland

ABSTRACT

In the computing education community, discussion is growing about the benefits of teaching programming by explicitly using notional machines to help students. To-date most work is largely theoretical and little work addresses actually using them in a classroom. This paper documents our experience of creating a notional machine for a specific course and using it in that classroom. A key point we learned while creating this notional machine is that many of the difficulties encountered were due to the concept of a notional machine being tightly coupled to students' mental models. Although not surprising, the numerous complications this brings are important to overcome. The potential amount of detail included in the notional machine is enormously influenced by the students' mental models, which are likely specific to a course, and also change throughout a semester – and certainly across several semesters. We present lessons learned from this experience, among them that implementing a notional machine and using it in class is a non-trivial yet possibly beneficial exercise.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; *CS1.*

KEYWORDS

code tracing; code writing; CS1; introductory programming; memory diagrams; mental models; notional machines; pedagogy; program construction; stack traces; visualization

ACM Reference Format:
Paul E. Dickson, Tim Richards, and Brett A. Becker. 2022. Experiences Implementing and Utilizing a Notional Machine in the Classroom. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499320

1 INTRODUCTION

It is generally accepted that programmers do not construct programs to be executed on an actual concrete computer about which they know all necessary detail. Instead they rely on abstractions and mental models which are often flawed, particularly for novices [19].

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03.
https://doi.org/10.1145/3478431.3499320

This direct path to hardware that does not actually occur is represented by the dashed line in Figure 1. Programs are instead constructed to be 'executed' on mental models of how the computer runs a programming language. In this context, the concept of a notional machine – an abstract computer responsible for executing programs of a particular kind [23] is helpful. However, this structure introduces several difficulties. The chief difficulty is that the notional machine must by definition be correct [23] but a students' mental model of the notional machine often is not. However, that correctness is the power of a notional machine. Further, the students' mental model is (through learning) continually changing. As Du Boulay, O'Shea & Monk stated: "Novices start programming with very little idea of the properties of the notional machine implied by the language they are learning" [9, p237].

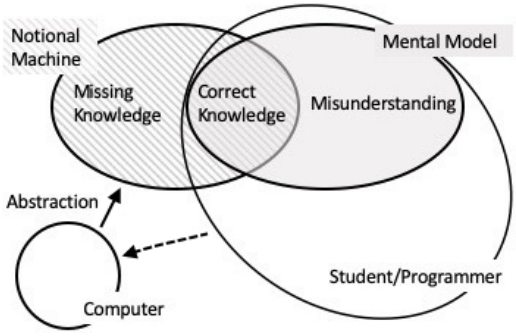

Figure 1: Relationship among the students, their mental model, a notional machine, and a physical computer (from [6]).

Other difficulties surround the notional machine itself: What exactly is a notional machine? How is it represented? What does it look like? The answers to these questions, based on the literature, is that as a community we are not entirely sure. Even the definition of a notional machine seems to subtly (but importantly) change from author to author. When the concept of a notional machine was introduced, du Boulay et al. said: 'The notional machine is an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed" [9, p237]. So a notional machine is an abstraction of a concrete computer that is consistent with it but omits much of the low-level details. Much of the current discussion of notional machines centers around notional machines being defined by a series of rules that provide an understanding of run time dynamics (execution) of a program [23]. Fincher et al. describe a notional machine as "a pedagogic device to

assist the understanding of some aspect of programs or programming" [10, p502]. These descriptions may not be at odds but, it is possible to imagine interpretations where they may be.

We talk about *a* notional machine and not *the* notional machine unless we are referring to a specific notional machine. *A* notional machine is *some* notional machine. This is because theoretically there are an infinite number of different notional machines for a given computer/language combination. For more on our understanding of what a notional machine is see [6]. We began this project to see if we could build a notional machine for a Python based CS1 course to both improve our students' learning and push forward our understanding of what a notional machine is. Our goal was to create a notional machine in order to give the community a starting point from which to begin shaping exactly what a notional machine for use in the classroom might look like.

Briefly, our interpretation of a notional machine in this context is that a person should be able to use a mental model to run code on a notional machine and achieve the same results as those achieved when the code is run for real, on the actual computer. Our goals were to define a notional machine in terms of rules and state. We represent state using hand-drawn memory diagrams that make it easy to see what information is stored and how. Our rules define how these diagrams are updated by different code constructs.

While writing the rules for our notional machine, we discovered that a notional machine and the concept of a mental model are so tightly intertwined that we could not define a rule without understanding the state of the mental model of the intended audience. For instance, at the start of a typical introductory programming course such as CS1 [2], a notional machine must include rules that explain how variables work. If one were to build a notional machine for CS2, the rules for defining variables would be written with less detail as students' mental models would be expected to include consistent variable concepts (such as one variable represents one value at one address). Thus, any explicitly constructed notional machine is partly defined by the mental models of the audience, which blurs the lines between a notional machine and a mental model, apart from the possibility of the mental model being flawed. If the argument is made that the notional machine should be independent of the mental model and that all possibly necessary details need be included, one realistic conclusion is that such a notional machine approaches being indistinguishable from formal definitions of the programming language and machine themselves. This, besides being impractical, defeats the purpose of notional machines entirely, at least for use in the classroom. After all, by definition a notional machine, and any usable mental models, are abstractions.

When attempting to use a notional machine in class, we ran into logistical issues made more complex due to the remote nature of the course due to COVID-19. Student responses to the idea of the notional machine were positive but they did not interact with our notional machine often during the semester. Many students commented that they would have looked at it more during class if they had enough screen area to view it and other activity simultaneously. Despite this, there are lessons learned that give insight into explicitly using a notional machine in class.

In this experience report we discuss building a notional machine for a specific CS1 course and the considerations that went into our design. We discuss questions this process raised regarding the practical definition of a notional machine such as where the lines are between a notional machine and mental models. We also discuss how our notional machine addresses specific common programming misconceptions of novice programmers. Finally, we discuss our experience using our notional machine in the classroom.

2 RELATED WORK

The belief that programming is hard seems widespread among teachers and researchers [1]. Difficulties surrounding mental models and notional machines may be partially responsible for this. It is also generally accepted that learning to program could be easier for many students [15]. This can at least partially be attributed to the tools and techniques used for teaching [14]. It is possible that properly used, notional machines may aid teaching and learning [10]. Approaches used to teach introductory programming have included the use of mental models and notional machines [10] however it is not common practice [17]. The notional machine literature is growing and relatively consistent – inasmuch as a recent works discuss most work to date, including [6, 8, 9, 12, 22]. Much of this work stems from observations that students should be helped in developing accurate mental models of a notional machine for the language they are learning, yet in practice novices often construct mental models inconsistent with actual computer behavior [16, 19]. In particular, "students commonly lack a viable model of program execution" [24, p179]. Examples of notional machines explicitly being used in the instruction of novice programmers is rare in the literature. Many papers discuss notional machines but do not develop and use them explicitly.

Berry & Kölling designed and implemented a notional machine similar to ours. It represents state using a graphical notation and rules to update that notation [3]. No formal user studies were completed and they do not appear to have explicitly described a notional machine to their students. Sorva & Seppälä introduced a framework called *principle of explicit program dynamics* that focused students on a continuous and explicit theme of run-time dynamics [26]. On the basis that a notional machine of some sort is present in any introductory programming course, although often left implicit, they provide an implementation-agnostic approach to making program dynamics more explicit to learners. The approach focused on concepts including the following: *terminology* that makes clear how words like 'program', 'dynamic', and 'static' are used; *explicit state* showing that a selection of transliminal concepts can bring students to new perspectives and therefore a focus on which program dynamics are necessary; and *"low"-level program visualization* that introduces their notional machine in terms of concepts such as memory and the stack. These concepts make execution more tangible while also providing a vocabulary to describe it. They advocate for such visualizations to be consistently used so that new concepts can be integrated. Although no rigorous evaluation was carried out, the authors noted a high pass rate and good student feedback.

Dickson et al. also advanced the benefits of explicitly using notional machines in the introductory programming classroom [6]. They advocated for a clear delineation between program visualization and the notional machine, using memory diagrams and other

means of visualization as examples to demonstrate these advantages. However this work was theoretical and did not report on using these techniques with students.

3 BUILDING A NOTIONAL MACHINE

Noting that studies of constructing and using notional machines with students are scarce, our goal was to see whether we could build and use a notional machine for the CS1 course taught in Python at the institution of one author. This provided us with a known set of materials and student knowledge for developing a notional machine. We hoped to guide students through the process of building a notional machine over the course of the semester. Thus, students would have ownership of the notional machine as they would have put concepts into words more accessible to them than the precise words of an instructor alone.

3.1 The Process

A notional machine is generally considered to be a set of rules that govern how information stored by the computer can be manipulated. They help provide an understanding of how the programming language works without unnecessary hardware-level detail. We began by taking the teaching notes for our CS1 course in Python and converting them into the rules needed to understand the programming concepts required. We concluded that more structure than just rules would be needed. Therefore, we broke our notional machine into three conceptual categories: *state*, *rules*, and *knowledge*. We view this as critical in creating a usable notional machine.

State is the information stored in memory and how it is stored. We decided to use Ithaca Memory Diagrams (IMDs) [7] for this. Using a graphical representation to embody state is an alternative to the view that a visualization of state is separate from the notional machine (see [11, p32] for a discussion). We agree that a visualization that is just a picture of state is not part of a notional machine. The important facet of IMDs is that they make it possible to describe state and not necessarily that they are visual. This appears similar to the interpretation used by Berry and Kölling [3] when creating their notional machine. We consider IMDs to be a method for keeping track of state and therefore include them as part of our notional machine definition. That is, we can develop rules on how the state represented by IMDs is updated as a program executes.

Rules are explanations of how the notional machine should update state or proceed based on different control structures in the code. While attempting to write rules to cover concepts governing program flow, we determined that flowcharts appear to have more potential for describing program flow than written explanations, at least for our pedagogical purposes. Following this we augmented our notional machine rules with flowcharts. Our rules use a combination of written explanations and flowcharts to convey concepts. No formal structure exists as to whether one or both methods of writing the rules is used, it is always whatever method is clearest for presenting the concept in question.

We developed a simplified flowcharting method similar to those previously used in computing education [4, 13, 21] but simpler. We removed much of the complexity inherent in others (both historical [5] and more modern incarnations like UML activity diagrams [18] and Drakon [28]) while conveying information in a form that addresses many common misconceptions typical of programming students. We were deliberate in doing so as the scope of our flowcharts is fairly narrow since we are not seeking to represent any possible program, only discrete programming concepts required for CS1. However, we have no reason to believe that our minimized flowcharts could not be easily extended with additional constructs that could be used in more advanced programming courses.

Knowledge is the information necessary to understand the rules but which did not fit cleanly into written rules, flowchart rules, or state. This knowledge is not a part of any definition of a notional machine found in the literature, but we found that it made the abstraction of the computer clearer. For example, the concept that information can be stored using a combination of symbols in the form of a variable is best called knowledge, whereas the statement that a variable is assigned the value on the right of an assignment operator is a rule. Our thought process was that knowledge would cover concepts while rules would cover specific applications of concepts. Although knowledge can possibly be written as rules and vice-versa, we believe that two separate categories make the structures more digestible. We also believe that when students attempt to use their mental models to run code on the notional machine, they have learned the knowledge part but possibly need to reference rules for specific situations. Our initial experience using our notional machine in the classroom did not provide enough evidence to suggest whether such assumptions were correct.

3.2 Details of State

As mentioned above we use IMDs to represent state. Visualizations like IMDs and flowcharts are not typically considered to be a part of a notional machine although they are often associated and used with them. Outside of notional machines, a large body of work exists on the benefit of using memory diagrams to teach and we do not delve deeply into that literature here. The reader is guided to Sorva [25]. One point made by Sorva is that for students to benefit from memory diagrams, they need to participate in their creation or development and not just passively observe them. This suggests that the hand-drawn representation of state provided by IMDs are ideal for students to use while running code on a notional machine. Our rules specify how IMDs are updated as a program executes.

When novices are introduced to the concept of variables, they often have a misconception that a simple variable can hold multiple pieces of information (Swidan et al. [27]). By using IMDs we can define variable behavior with the following rules that update the IMD and encapsulate this knowledge. These rules relate to the diagram and code found in Figure 2.

- When the program encounters a variable block, it creates a name and a value in the memory diagram.
- When a variable block assigns to a variable that already exists, the original value is crossed out and replaced with the new value; the old value is gone.

The IMDs also specifically address each of the following misconceptions common to CS1 students (from [27]) by showing exactly how code affects memory in each situation based on the rules that make up our notional machine.

M9 A variable can hold multiple values at a time (the second rule in the previous list addresses this).

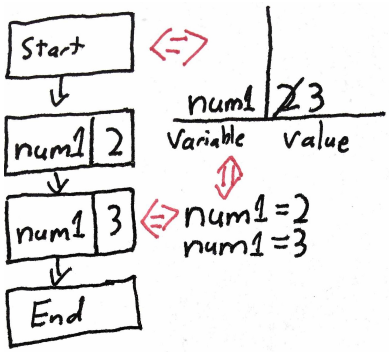

Figure 2: Variable definition in an Ithaca Memory Diagram.

M14 A variable is a pairing of a name to a changeable value. It is not stored inside the computer. (IMDs are specifically a representation of what is stored in the computer's memory.)

M15 Primitive assignment stores equations or unresolved expressions. (Only values can appear to the right of the variable name in an IMD.)

3.3 Details of Rules

Determining how to write the rules associated with the notional machine proved the most difficult part of defining it. One major difficulty was determining the level of specificity to use. This issue kept arising as we either felt that we had not covered all contingencies or if we did, we were approaching language-level specification. This brought us back to our intended approach: having students be an integral part in the notional machine creation as the semester progressed. We thought that if the students would eventually write the rules in their own language (and not the more nuanced and precise but sometimes less clear language that instructors use), they would write it with the correct level of detail for their mental models. This has the benefit of putting the notional machine into language that is more accessible for our students as it is in their language from the start. This approach was applied to the knowledge category also for the same reason. One of the challenges of this plan is to make sure that what the students are creating is in the form of rules and knowledge and not just informal course notes. If we could properly shape these sets of rules and knowledge, we may eventually arrive at a more usable notional machine. We might even be able to drop the knowledge category, which would become part of students' mental models – arguably one definition of learning provided the knowledge is correct.

As discussed, we came to believe that using flowcharts and their associated rules when possible would be more intuitive for students. This parallels how we previously used flowcharts to discuss concepts such as *if* statements and *while* loops in class. For the notional machine we have done our best to standardize the flowcharts in order to make them a consistent part of the overall notional machine rules. In our flowcharts, diamonds are decision nodes, capital letters are boolean expressions, rectangles are execution nodes, lowercase letters are executable code, and letters are concatenated when a particular location is only reachable through a specific branch path. Letters do not imply any ordering (e.g. alphabetical).

Our first example of rules relates to *if* and *elif* without any *else*. Figure 3 shows the program flow for this example. The paths through the code illustrate execution flow without trying to put it into words – at least not initially. This addresses two more of the common misconceptions from Swidan et al. [27]:

M23 Difficulties in understanding the sequentiality of statements.

M26 A false condition ends program if no *else* branch exists.

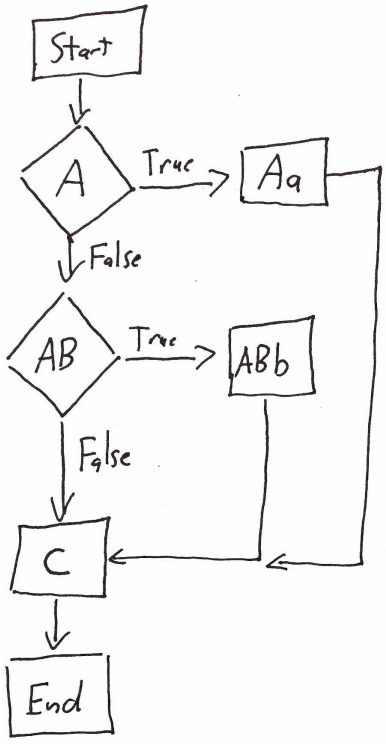

Figure 3: flowchart rule for *if/else if* statement.

Flowcharts with rules also make it easy to show the cyclic nature of a *while* loop as shown in Figure 4. We believe the picture is an easier way to follow and understand control flow than an explanation in words. However, we recognize that the diagram by itself is not enough and therefore extended our notional machine with additional clarification:

- Knowledge: A *while* loop is similar to an *if* statement in that both chose one of two arrows based on a boolean expression, they differ in that a while loop arrow can go back to a diamond above that includes a match of the left-most uppercase letters.
- Rule: Arrows can go back to previous decision nodes but only if those nodes are in the path followed.
- Rule: Arrows back cannot point to a node that has a uppercase letter lowercase letter immediately off the false branch.

This exemplifies how we built and refined the notional machine. We believe that examples like these help eliminate two more common misconceptions from Swindan et al. [27]:

M31 Control goes back to start when condition is false.

M33 Loops terminate as soon as condition changes to false.

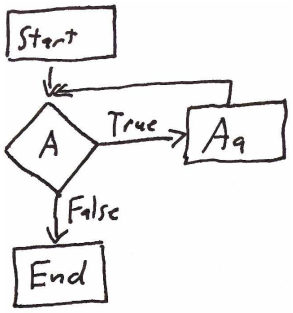

Figure 4: flowchart rule for *while* loop.

3.4 Details of Knowledge

Although the knowledge category is beneficial as part of our specific notional machine design, it raised the most questions about where the notional machine ends and the mental model begins. The purpose of the knowledge category is to contain information that must be known for the rules to be applied. An example of knowledge from our notional machine is "Python contains a *boolean* type of data that can be *True* or *False*." Additionally, this type is used in conditional statements. Although this could be written as a rule, it is clearer as simply stated (and hopefully assumed) knowledge.

A more controversial piece of knowledge is "Math's order of operations hold true." It is easy to write out the order that operations are performed in as a series of rules, but to what benefit? Doing so would just create a longer and more detailed notional machine specification. The needed knowledge is the order of operations taught in math courses, and a corresponding rule would simply be to apply that knowledge. This relies on the mental model being able to tap into (hopefully existing) knowledge and defines the notional machine in terms of what the students know. In this way, the level of abstraction of the notional machine is defined by the students' mental models (and those models, by the course content). Therefore the notional machine should change with time. The knowledge in our notional machine from the start of the semester was:

- A program can have variables.
- A variable is how the computer remembers data.
- A variable can hold 1 value at a time.
- A variable can change the value it stores through assignment.
- In Python variable names can include numbers and letters but must start with a letter, and no spaces can be a part of the name.
- To set a variable the name must be on the left of the assignment (equal) operator.

By the end of CS1 only the last two pieces of knowledge should be necessary as the first four should be firmly part of the mental model. While the notional machine could still contain this level of detail, it would return to the question of whether the notional machine needs to be so detailed that it ceases to be an abstraction and is more akin to a language specification in another form.

4 NOTIONAL MACHINES IN THE CLASSROOM

During the Fall 2020 semester we attempted to incorporate our notional machine into the CS1 course for which it was designed. The intention was to teach the course using a combination of a computer to write code and whiteboards for memory diagrams and flowcharts. Students were intended to have laptops open to follow along and try code while also keeping a shared Google document open that would hold the class notional machine. Students would also have paper to copy down IMDs and would later incorporate these IMDs into the class notional machine.

That semester campus was operating remotely due to COVID-19. Students were given a link to a Google document where the class notional machine would be documented and were shown how it was intended to be updated. Over the first couple of classes, the author/instructor updated the document during class in order to prime the pump for students. After three weeks during which students barely modified the document, the instructor decided to update the class notional machine for each new concept presented using the material from the notional machine built before the semester. It was believed at the time that students simply did not have enough knowledge to feel comfortable updating the notional machine and that giving them a starting point to modify would be more effective. Despite this change, students never took ownership of the notional machine and did not modify it to any great extent. As the semester went on the instructor continued to update it but it stopped being emphasized as students had not bought into the concept and it was taking too much class time to update.

At the end of the semester a survey was conducted that asked about the notional machine and flowchart aspects of the class. A discussion was also held where students were able to express their opinions without worry of retribution. It is worth noting that students were informed of the experimental nature of the notional machine at the start of the semester; when changes were made, they were informed as to why the change was being made.

The survey and discussion showed that students were generally interested in the idea of the notional machine and the use of flowcharts. In the discussion one student stated that they had started to ignore the notional machine document because they simply did not have the screen real estate to keep it open during class with so much being needed for the class Zoom meeting and Python editor. Updating the notional machine afterward was just one too many things to do in a class that already featured a heavy workload. The rest of the students largely agreed with this sentiment.

Students commented that the notional machine would have been more helpful if they had dedicated time to update it during class. They expressed that it was useful to try to clear up misconceptions. Considering that the notional machine document was the only reference they had for the specific flowcharts used in class, the question must be raised as to whether the notional machine was being used as such, or simply as a good set of notes to learn and improve mental models. This also begs the question as to where the boundaries are between notional machine, visualization, mental model, and course notes when using a notional machine in class.

5 BENEFITS

Research on notional machines suggests that they have the potential to aid student learning. We described our experience in developing a specific notional machine that addresses some common student misconceptions, but notional machines defined differently from ours

could also achieve the same goals. An added benefit was that we were able to demonstrate certain aspects of notional machines that were previously assumed but not elaborated upon. One example is that it was assumed that the notional machine would help students see similarities between programming concepts without getting bogged down in language-specific syntax. The figures in Section 3.3 show examples of *if* statements and *while* loops described by rules that are syntax independent, and provide a specific example of this language independence. This was witnessed in class assignments and assessments in which students were able to draw flowcharts that showed correct program flow to solve a problem while being unable to write correct program syntax for the same problem. The IMDs and how they are used to store state information also work independent of a specific programming language, showing how the concepts of state can hold across languages.

Another benefit mentioned above is that by allowing students to help shape the notional machine, we enable them to put computing into the language of their own backgrounds and the context of their own knowledge. The language of computing instruction is typically precise to avoid ambiguity, but for many of our students, especially those without strong backgrounds, this can be confusing. It can be easy to lose students in attempts to be formal and unambiguous. We believe that allowing students to build a class notional machine during the semester, the notional machine will be in their own words and reflect their backgrounds, making the subject matter more accessible to a greater variety of students.

6 DISCUSSION

Specifying a notional machine was more difficult than we thought it would be. The thought process for determining how any notional machine must reflect users' mental models is complicated. While an argument can be made that it is possible to create universally accessible rules for a notional machine without regard for the mental model, we do not agree in practice. This sentiment is reflected in the literature at times although not often overtly. For example, the rules of Berry and Kölling [3] require a certain level of understanding in order to be used (i.e., knowing what an object is). A rule covering a concept as basic as a variable storing data would be written differently based on whether the mental models of those who will use the notional machine have seen variables in algebra.

This leads us to the question of whether it is possible to have a notional machine without a specific mental model. Without getting too far into philosophy, we do not believe that to be useful. We believe that a notional machine can be written for an assumed baseline mental model for the intended audience. For example, a notional machine for a specific class of students can be written assuming the baseline knowledge on the basis of prerequisite courses. While some students will have more information in their mental models and therefore not need certain pieces of the notional machine to be included, the extra detail is not detrimental if level-appropriate.

Our experience leads us to believe that a notional machine and mental model are to a certain extent inseparable. It is possible, however, to approach this from other angles, and counterarguments for some of our points are possible. Nonetheless we did strive for practicality first and foremost – a goal that does not come without some sacrifices. Our question of whether explicitly using a notional machine as part of an introductory programming course is beneficial was not fully answered. Our experience did not allow us to test this. Students successfully used IMDs and flowcharts throughout the course and benefited from them but it is unclear whether the notional machine explicitly provided benefit. The fact that students seemed to primarily use the notional machine as notes to help understand concepts and solve problems brings the question of what a notional machine is back to the forefront. How is a notional machine different from a correct set of class notes that a student can use in order to accurately predict the outcome of required code? Is a notional machine at its heart really just a concise set of notes on a programming language that defines how that programming language works in enough detail for the user's mental model, in a given context?

The faculty members who developed this instance of a notional machine benefited from creating it as the need for consistency led to a clearer method of organizing course material which led to a better course design overall. We feel it also gave us more insight as to what a notional machine is and how they might benefit students.

The area of notional machines, like other areas of computing education research, have few (or no) common standards for how data and artifacts should be represented, or shared [20]. We have published our notional machine specification[1] for others to use.

7 CONCLUSIONS

In this paper we document the process of developing a notional machine for a specific course and deploying it in class. Our notional machine is not perfect. Part of the imperfection of our notional machine is that many concepts did not fit neatly into our categories of *state*, *rules*, and *knowledge*. Some concepts could indeed be addressed by more than one of these. Our experience leads us to believe that this may be unavoidable. Also, a lot of details are not explicitly included in our notional machine although they are part our CS1 course. For example, string manipulation, which is really just a new use of existing rules, may seem like new concepts to students – they need to see that the existing implementation can be used to address such concepts although this may not be explicitly apparent. Despite this we believe that we have developed a notional machine with an appropriate level of abstraction and detail for our CS1 course. We believe that we have made a compelling argument for one possible way of explicitly incorporating notional machines into a classroom environment even if we were not able to back this up with much classroom evidence.

Our concrete example of a notional machine could help answer the question of what a notional machine for use in the classroom might look like. It gives a perspective on how mental models, notional machines, program state, program visualizations, knowledge, and course notes are related. In our notional machine the mental model provides the context within which the rules of the notional machine are written. A visualization is a snapshot of the current state, which is the result of the application of the rules. All of these seem to be required to provide an abstraction upon which students can better understand what code is and how it works.

[1]For the full notional machine that covers all of our CS1 concepts including functions and methods please visit https://doi.org/10.5281/zenodo.5760324

REFERENCES

[1] Brett A. Becker. 2021. What Does Saying That 'Programming is Hard' Really Say, and about Whom? *Commun. ACM* 64, 8 (jul 2021), 27–29. https://doi.org/10.1145/3469115

[2] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432

[3] Michael Berry and Michael Kölling. 2014. The State of Play: A Notional Machine for Learning Programming. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 21–26. https://doi.org/10.1145/2591708.2591721

[4] Dennis J. Bouvier. 2003. Pilot Study: Living Flowcharts in an Introduction to Programming Course. *ACM SIGCSE Bulletin* 35, 1 (Jan. 2003), 293–295. https://doi.org/10.1145/792548.611991

[5] Ned Chapin. 1970. Flowcharting With the ANSI Standard: A Tutorial. *Comput. Surveys* 2, 2 (June 1970), 119–146. https://doi.org/10.1145/356566.356570

[6] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 159–165. https://doi.org/10.1145/3341525.3387404

[7] Toby Dragon and Paul E. Dickson. 2016. Memory Diagrams: A Consistant Approach Across Concepts and Languages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 546–551. https://doi.org/10.1145/2839509.2844607

[8] Benedict du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[9] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies* 14, 3 (1981), 237–249.

[10] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Capturing and Characterising Notional Machines. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 502–503. https://doi.org/10.1145/3341525.3394988

[11] Mark Guzdial. 2015. Learner-centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics* 8, 6 (2015), 1–165.

[12] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019), 1–23. https://doi.org/10.4230/DagRep.9.7.1

[13] Richard E. Haskell, David E. Boddy, and Glenn A. Jackson. 1976. Use of Structured Flowcharts in the Undergraduate Computer Science Curriculum. *ACM SIGCSE Bulletin* 8, 3 (July 1976), 67–74. https://doi.org/10.1145/952991.804758

[14] Ioannis Karvelas, Joe Dillane, and Brett A. Becker. 2020. Compile Much? A Closer Look at the Programming Behavior of Novices in Different Compilation and Error Message Presentation Contexts. In *United Kingdom & Ireland Computing Education Research Conference. (UKICER '20)*. Association for Computing Machinery, New York, NY, USA, 59–65. https://doi.org/10.1145/3416465.3416471

[15] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 759–765. https://doi.org/10.1145/3328778.3366882

[16] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Chapter 13, 377–413. https://doi.org/10.1017/9781108654555.014

[17] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[18] (OMG) Object Management Group. 2020. https://www.omg.org/spec/UML/

[19] James Prather, Brett A. Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What Do We Think We Think We Are Doing? Metacognition and Self-Regulation in Programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3372782.3406263

[20] Thomas W. Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M. Kazerouni, Brett A. Becker, Andrew Petersen, Luke Gusukuma, Stephen H. Edwards, and David Babcock. 2020. ProgSnap2: A Flexible Format for Programming Process Data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 356–362. https://doi.org/10.1145/3341525.3387373

[21] H. Rudy Ramsey, Michael E. Atwood, and James R. Van Doren. 1983. Flowcharts versus Program Design Languages: An Experimental Comparison. *Commun. ACM* 26, 6 (June 1983), 445–449. https://doi.org/10.1145/358141.358149

[22] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education; Visuaalinen Ohjelmasimulaatio Ohjelmoinnin Alkeisopetuksessa*. Ph.D. Dissertation. Aalto University, Espoo, Finland. http://urn.fi/URN:ISBN:978-952-60-4626-6

[23] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[24] Juha Sorva. 2018. Misconceptions and the Beginner Programmer. In *Computer Science Education: Perspectives on Teaching and Learning in School*, Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). Bloomsbury Academic, Chapter 13, 171–188.

[25] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4, Article 15 (Nov. 2013), 64 pages. https://doi.org/10.1145/2490822

[26] Juha Sorva and Otto Seppälä. 2014. Research-based Design of the First Weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2674683.2674690

[27] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 151–159. https://doi.org/10.1145/3230977.3230995

[28] Andrey Terekhov, Timofey Bryksin, and Yurii Litvinov. 2017. *How to Make Visual Modeling More Attractive to Software Developers*. 139–152. https://doi.org/10.1007/978-3-319-67425-4_9