TRACESHEETS
Spreadsheets of Program Executions as a Common Ground between Learners and Instructors

Soichiro Fujii and Hisao Tamaki
Department Computer Science, School of Science and Technology, Meiji University, Tama, Kawasaki, 214-8571, Japan

Keywords: Tracesheet, Spreadsheet, Introductory Programming Course, Tracing Program Execution, Debugging.

Abstract: We propose the use of spreadsheets representing program executions in introductory programming courses. Each row of a spreadsheet is a snapshot of the execution at each time step showing the values of variables and each column corresponds to a particular variable showing the entire history of its updates. We call such a spreadsheet a tracesheet. We discuss the motivation and potential benefits of the use of tracesheets in introductory programming courses, discuss some design issues, and report on a preliminary implementation of a tracesheet generator for Java.

1 INTRODUCTION

It is observed in introductory programming courses that insufficiently performing students tend to view a program as a static textual object rather than a dynamic object that generates a process. Most of them are capable of step-by-step tracing of a code but exercise this ability only when they are forced. Thus, their preferred way of determining the correctness of a code is to compare it with what they believe is a correct code: one given in the textbook, on the blackboard, written by a reliable friend, or posted on the internet. These tendencies are severe obstacles not only for debugging but also for the entire process of leaning to program.

Based on this observation, we believe that the research on programming education should focus not only on how to teach each subject in most understandable manners but also on a long-term methodology and tools for remedying the above mentioned undesirable tendencies of some students and enhancing their capability and willingness to use step-by-step tracing for understanding a code. One might hope that extensive uses of program animation (Moreno et al., 2004), (Levy et al., 2003), (Sajaniemi and Kuittinen, 2003), (Sutinen et al., 2003), (Lahtinen et al., 1998), (Baecker, 1998), (see (Urquiza-Fuentes and Ángel Velázquez-Iturbide, 2009), (Shaffer et al., 2010) for a recent survey) would serve this purpose well. Program animators, however, are designed for different purposes. They are meant to help students understand a code by *giving* a graphical and often conceptual view of the behavior of the code, rather than encouraging the students to develop such a view themselves from the given code. Thus, it is possible that a student perfectly understands how a code operates through the given visualization and yet is incapable (or more likely reluctant) of tracing a similar or even the same code when left on his/her own. While the program animators could be adapted for our present purposes, for example by engaging students (which is getting recognized as indispensable in effective uses of program animation (Hundhausen et al., 2002), (Urquiza-Fuentes and Ángel Velázquez-Iturbide, 2009)), we pursue a different approach in this paper.

Based on experiences in introductory programming courses, we believe in the need of treating the raw, unabridged, and unstructured description of a program execution as a first class object that is *explicitly* shared by instructors and students. When a program code is shown in traditional programming courses, the precise execution process specified by the code is usually implicit. The instructors verbally summarize the execution of the code and only occasionally spell out the full details of the execution. Here we recognize a serious gap between the instructor and students: they are looking at the same code but may not be seeing the same thing. For the instructors, mentally generating the process behind the code is easy and automatic, so in a sense they see the process whenever they see a code. On the other hand,

Fujii S. and Tamaki H..
TRACESHEETS - Spreadsheets of Program Executions as a Common Ground between Learners and Instructors.
DOI: 10.5220/0003916701580163
In *Proceedings of the 4th International Conference on Computer Supported Education* (CSEDU-2012), pages 158-163
ISBN: 978-989-8565-06-8

for beginning students, the same mental activity is demanding if at all possible and the code they are looking at may remain a purely textual static object in their view.

Since the instructors' view of the process behind the code is structured, they explain the code based on this structure. This itself is a right approach but, in doing so, they often forget the fact that their ability to see the structure comes from their long experience in which they dealt with the raw execution of the code. Since the structure of the execution process reflects the structure of the code, the instructors expect that students accept their explanations without any difficulty – our experiences show that this expectation is not always fulfilled. We ascribe the cause of this failure to the above mentioned gap between the views of the two parties: since the students do not have the basis (the raw execution) in their mind to build the structure on, the structure given to them can be abstract, fragile, and intangible.

In this paper, we propose the use of spreadsheets to explicitly represent the raw, full detailed execution of codes. Each row of the spreadsheet represents a program state at each time step in terms of the values stored in the variables and each column represents the changing values of each variable. We call this representation a *tracesheet*. Such representations are quite natural but, to the best of the present authors' knowledge, no report of their use in programming education is found in the literature. We are, however, inspired by the use of spreadsheets in teaching logic design and computer architecture (Timsit and Zertal, 2010).

First of all, we have to emphasize that a tracesheet is *in no way intended for visualization*. Although it is 2-dimensional, it is basically textual and as the number of variables and the number of execution steps increase, it becomes painful to read its entries. It is simply an explicit representation of an awkward existence, which is usually left implicit, which both the instructors and students prefer to stay away from, but which yet is to be closely examined by both parties in crucial occasions. It is meant to pave the gap between the instructors' and students' views discussed earlier: by looking at the same explicit representation of the code's behavior, the chances of both parties seeing the same thing would be increased.

In closing this introduction, we compare conventional debuggers with tracesheets in the context of introductory programming education. It is true that such debuggers can provide all the information on a tracesheet. They are, however, not meant to present such information in their entirety. Rather, they are designed to help the user navigate through the massive body of information unsuitable, if not impossible, to be put on a single sheet. For such navigations to be successful, the user must have sufficient knowledge of programming, of debugging and of the particular debugger being used. For small programs that appear typically in introductory programming courses, the entire description of program execution on a tracesheet is manageable through the simple user interface of scrolling and has a definite advantage for beginning students that no additional knowledge of the tool is required.

The rest of this paper is organized as follows. In Section 2, we illustrate the possible uses of tracesheets with a few examples. In Section 3, we discuss issues to be considered in the design of tracesheets and describe our decisions. In Section 4, we describe our preliminary implementation of a tool that generates a tracesheet from a Java source code.

2 EXAMPLES

In this section, we illustrate the use of tracesheets by some examples.

Our first example is a simple for-statement. Figure 1 shows a tracesheet generated from the following code.

```
int n = 3;
int s = 0;
for (int i = 0; i < n; i = i + 1) {
  s = s + i;
}
```

Instructors can use this tracesheet in a tutorial introducing for-statements. As they verbally describe the behavior of the code, they can point to or highlight the particular entry in the sheet corresponding to the event they are referring to. While following the instructors' description, students are also able to resolve many small questions they ask themselves, based on the entire scenario of execution shown on the sheet.

In our implementation of tracesheets, the initial values of variables can be edited on the sheet so that the sheet dynamically changes depending on the edited values (see Section 3). Thus, for example, after describing the behavior of the code and the roles of variables, the instructors may announce that he will change the initial value of n to 5, ask the students to predict the result, and then show the changes on the sheet for confirmation. Changing the initial value of s or i may also be instructive. If the tutorial is done in a lab, then the tracesheet can be distributed to the PCs of students and students can play with this feature on their own until they have full understandings of the behavior of the code and its dependence on the initial values of the variables.

	A	B	C	D	E	F
1	no	code	cond	n	s	i
2	0			?	?	?
3	30	int n		?	?	?
4	31	=3		3	?	?
5	40	int s		3	?	?
6	41	=0		3	0	?
7	51	int i		3	0	?
8	52	=0		3	0	0
9	53	i < n	TRUE	3	0	0
10	60	s=s + i		3	0	0
11	54	i=i + 1		3	0	1
12	53	i < n	TRUE	3	0	1
13	60	s=s + i		3	1	1
14	54	i=i + 1		3	1	2
15	53	i < n	TRUE	3	1	2
16	60	s=s + i		3	3	2
17	54	i=i + 1		3	3	3
18	53	i < n	FALSE	3	3	3
19	70	end		3	3	3

Figure 1: Example 1: a simple for-statement.

Tracesheets can be used not only in tutorials but also in a more personal session between a student and a (human or automated) instructor. Suppose that a student submitted the following incorrect code for the problem of assigning the maximum value of *a*, *b*, and *c* to variable *max*.

```
max = a;
if (b > a) {
  max = b;
}
if (c > a) {
  max = c;
}
```

When the instructor explains when this code goes wrong, a tracesheet for such a case, with say $a = 1$, $b = 3$, and $c = 2$, (see Figure 2) would definitely help the student to grasp the problem. If the emphasis of the session is more on letting the students discover the bugs on their own, then the instructor may provide the student with the tracesheet with initial values say $a = 1$, $b = 2$, and $c = 3$, for which the code achieves the correct result, and ask the student to experiment on the values to find a case in which the code fails. Although such an experiment can also be done with the code itself, the task of finding the *cause* of the incorrect result would be much easier on the tracesheet.

We end this section with an example involving an array. Figure 3 shows a tracesheet generated from the following incorrect code for selection sort, again submitted by some student.

```
for (int i = 0; i < n; i++) {
  int m = i;
  for (int j = 0; j < n; j++) {
    if (a[j] < a[m]) {
      m = j;
    }
  }
  int w = a[i];
  a[i] = a[m];
  a[m] = w;
}
```

	A	B	C	D	E	F	G
1	no	code	cond	max	a	b	c
2	0			?	?	?	?
3	30	int max		?	?	?	?
4	31			0	?	?	?
5	40	int a		0	?	?	?
6	41	=1		0	1	?	?
7	50	int b		0	1	?	?
8	51	=3		0	1	3	?
9	60	int c		0	1	3	?
10	61	=2		0	1	3	2
11	80	max=a		1	1	3	2
12	90	b > a	TRUE	1	1	3	2
13	100	max=b		3	1	3	2
14	120	c > a	TRUE	3	1	3	2
15	130	max=c		2	1	3	2
16	140	end		2	1	3	2

Figure 2: Example 2: the maximum of three with a bug.

To be correct, the inner for-statement must start with $j = i$ or $j = i + 1$ rather than $j = 0$ as in the code. The entire execution history as given in the tracesheet may appear intimidating to the student at first. With suitable guidance of the instructor, however, the task of locating the problem on the tracesheet should not be so difficult. For example, the instructor may ask the student to find the first step at which the values in the arrays look wrong. A student with a clear abstract level algorithm in mind should be able to answer this question fairly easily. Then, the student will be asked to analyze the cause of the wrong value of *m* at that step. Many students will notice that the range of the inner for-statement must be corrected. Others will be encouraged to examine the previous steps that determined the value of *m*. If the student fails to answer the first question, then probably the discussions must be directed toward the idea of the algorithm at a higher level.

A more traditional alternative to the above scenario is to let the students trace the code themselves. Although this approach has its own advantages, the problem is that the task is too laborious for many beginning students. We may guide (or force) them, in the session, to trace the code and discover bugs, but our experiences show that most of them do not adopt this practice for their everyday use. Our approach is intended to make the task less laborious in the beginning phase. Given the tracesheet, the students can skip the laborious part and can concentrate on examining the trace. The hope is that, after seeing many traces explicitly given to them and experiencing successes in analyzing them, they will eventually be able, and willing, to generate traces on their own.

3 DESIGN ISSUES

3.1 Target Language and Subsetting

The idea of tracesheets can be applied to any proce-

	A	B	C	D	E	F	G	H	I	J	K	L
1	no	code	cond	n	i	m	j	w	a[0]	a[1]	a[2]	a[3]
7	61	int i		4	?	?	?	?	8	33	5	7
8	62	=0		4	0	?	?	?	8	33	5	7
9	63	i < n	TRUE	4	0	?	?	?	8	33	5	7
10	70	int m		4	0	?	?	?	8	33	5	7
11	71	=i		4	0	0	?	?	8	33	5	7
12	81	int j		4	0	0	?	?	8	33	5	7
13	82	=0		4	0	0	0	?	8	33	5	7
14	83	j < n	TRUE	4	0	0	0	?	8	33	5	7
15	90	a[j] < a[m]	FALSE	4	0	0	0	?	8	33	5	7
16	84	j++		4	0	0	1	?	8	33	5	7
17	83	j < n	TRUE	4	0	0	1	?	8	33	5	7
18	90	a[j] < a[m]	FALSE	4	0	0	1	?	8	33	5	7
19	84	j++		4	0	0	2	?	8	33	5	7
20	83	j < n	TRUE	4	0	0	2	?	8	33	5	7
21	90	a[j] < a[m]	TRUE	4	0	0	2	?	8	33	5	7
22	100	m=j		4	0	2	2	?	8	33	5	7
23	84	j++		4	0	2	3	?	8	33	5	7
24	83	j < n	TRUE	4	0	2	3	?	8	33	5	7
25	90	a[j] < a[m]	FALSE	4	0	2	3	?	8	33	5	7
26	84	j++		4	0	2	4	?	8	33	5	7
27	83	j < n	FALSE	4	0	2	4	?	8	33	5	7
28	130	int w		4	0	2	?	?	8	33	5	7
29	131	=a[i]		4	0	2	?	8	8	33	5	7
30	140	a[i]=a[m]		4	0	2	?	8	5	33	5	7
31	150	a[m]=w		4	0	2	?	8	5	33	8	7
32	64	i++		4	1	?	?	?	5	33	8	7
33	63	i < n	TRUE	4	1	?	?	?	5	33	8	7
34	70	int m		4	1	?	?	?	5	33	8	7
35	71	=i		4	1	1	?	?	5	33	8	7
36	81	int j		4	1	1	?	?	5	33	8	7
37	82	=0		4	1	1	0	?	5	33	8	7
38	83	j < n	TRUE	4	1	1	0	?	5	33	8	7
39	90	a[j] < a[m]	TRUE	4	1	1	0	?	5	33	8	7
40	100	m=j		4	1	0	0	?	5	33	8	7
41	84	j++		4	1	0	1	?	5	33	8	7
42	83	j < n	TRUE	4	1	0	1	?	5	33	8	7
43	90	a[j] < a[m]	FALSE	4	1	0	1	?	5	33	8	7
44	84	j++		4	1	0	2	?	5	33	8	7

Figure 3: Example 3: incorrect code for selection sort.

dural language. The choice of Java as our target language simply comes from a larger project involving the authors which is aimed at developing contents and support for leaning Java programming. Having chosen Java, we still have to choose a suitable subset for the tracesheets to cover. It is clear that we do not need the entire Java language as described in the language specification (Sun Microsystems, 1995) in introductory courses. Whether or not to include the following features is crucial in the design of tracesheets:

1. dynamic variable declarations,
2. method calls, and
3. objects.

Omission of all of these features would lead to a simple sheet layout and the induced subset of Java would still be useful in the very early stages of introductory courses. Our decision, however, is to include all of these three features. The reason of this decision is because method calls and objects are the main causes of students' difficulty in understanding the behavior of codes and we should at least try to see if our approach can be used to reduce the difficulty. A consequence of our decision is that our tracesheets must simulate both a stack, for allocating dynamic variables, and a heap, for storing objects. This somewhat complicates the layout of tracesheets (see below). For very beginning learners, we may need a separate basic mode in which variables are statically allocated to fixed columns, as in the examples used in Section 2.

3.2 Sheet Layout

Because of our decision to include dynamic variable declarations and objects, each column of a tracesheet corresponds to a memory cell rather than to a variable as in the examples in Section 2. Thus, columns are organized into a stack and a heap just as memory cells are organized into a stack and a heap in standard Java implementations.

Besides the columns corresponding to memory cells, we have three columns on the left of the sheet: the second column shows the code fragment executed in the step, the first column shows the number identifying the code fragment, and the third column shows the value of the boolean expression evaluated at the step if the expression is used as the condition in an if-statement or a for-statement.

	A	B	C	D	E	F	G	N	O	P	Q	R	S	T
1	no	code	cond	stack	stack	stack	stack	heap	heap	heap	heap	heap	heap	heap
2	0			?	?	?	?	?	?	?	?	?	?	?
3	30	int[] a		a	?	?	?	?	?	?	?	?	?	?
4	31	=[8,33,5,7,		N	?	?	?	6	8	33	5	7	11	3
5	50	int min		N	min	?	?	6	8	33	5	7	11	3
6	51	=a[0]		N		8	?	6	8	33	5	7	11	3
7	61	int i		N		8	i	6	8	33	5	7	11	3
8	62	=1		N		8	1	6	8	33	5	7	11	3
9	63	i < a.length	TRUE	N		8	1	6	8	33	5	7	11	3
10	70	min > a[i]	FALSE	N		8	1	6	8	33	5	7	11	3
11	64	i++		N		8	2	6	8	33	5	7	11	3
12	63	i < a.length	TRUE	N		8	2	6	8	33	5	7	11	3
13	70	min > a[i]	TRUE	N		8	2	6	8	33	5	7	11	3
14	80	min=a[i]		N		5	2	6	8	33	5	7	11	3
15	64	i++		N		5	3	6	8	33	5	7	11	3

Figure 4: Array allocation in heap.

The tracesheet shown in Figure 4, generated from the following code, illustrates how an array object is allocated in heap and how the object is referenced by a variable. In row 3 of the sheet, variable *a* is allocated in the stack. Then in row 4, an array object is created in the heap space, initialized as specified by

the initializer in the code. This array object is referenced by variable *a* by the column identifier N. Thus, the column identifiers serve as memory addresses in the sheet.

```
10  public class Sample {
20     public static void main(String[] args) {
30        int[] a = {8,33,5,7,11,3};
40
50        int min = a[0];
60        for(int i=1;i<a.length;i++) {
70           if(min>a[i])
80              min = a[i];
90        }
100    }
110 }
```

3.3 User Interaction

Tracesheets can be read-only: once they are generated, they do not change unless regenerated from a modified source code. However, it would be helpful for them to be interactive, as "engagement" is recognized as a vital factor in determining the effectiveness of programming education tools (Hundhausen et al., 2002), (Urquiza-Fuentes and Ángel Velázquez-Iturbide, 2009). We may expect a tracesheet to respond to

1. a change of the initial value of a variable, or
2. minor changes in a code fragment,

as the changes are typed directly into the sheet. We believe that both types of interactions are useful. The ease of implementation, however, differ substantially between the two types. The first type of interactions can be implemented without difficulty using the usual feature of spreadsheet applications: automatic recalculation of the entire sheet upon a change in a single cell. Implementing the second type of interactions using the same feature is not theoretically impossible but would be extremely complicated.

Moreover, we envision various uses of tracesheets in an interactive learning system where more sophisticated user interactions are necessary. For example, to learn the semantics of a new programming construct, filling in empty cells of a partially-filled tracesheet may be helpful. Another example is when a learner completes a programming task and submits the resulting program to the system. To confirm the understanding of the learner, the system may ask several questions regarding the behavior of the submitted program, where a tracesheet will be an appropriate media of communication. It is the topic of our ongoing research to decide what types of interactions are needed in such situations.

4 IMPLEMENTATION

In this section, we describe a preliminary implementation of a tracesheet generator for Java. Although the only frontend supported is Excel (Microsoft Corporation, 1985) in the current version, it is easy to adapt the output of the generating engine to any widely used spreadsheet applications or to custom-made frontends to be developed for more sophisticated user interactions as mentioned in the previous section.

The Java languages features covered by the current version include dynamic variable declarations, assignments, if-statements, for-statements, break-statement, and read-only arrays.@Of the primitive data types, only the "int" type is supported. The next version currently under development will cover general classes and objects, including array objects, and general methods.

In order to provide user interactions of type 1 as described in Section 3.3, the entry of each sheet cell is an Excel formula that evaluates to the value of the cell. In the subsequent subsections, we describe these formula entries and how they are generated from the given Java code.

4.1 Formula Entries

The value of a cell of a column corresponding to a memory cell is determined by the following:

1. the code fragment being executed, and
2. the values of memory cells in the previous step.

Thus, the formula for such a cell consists of a reference to the line number given in the first column of the same row and to the cells of the previous row corresponding to memory cells. For example, the formula entry in cell F10 of the sheet shown in Figure 4 is as follows.

```
=IF(A10=61,"i",IF(A10=62,1,IF(A10=64,F9+1,F9)))
```

If the current code fragment is the initialization of the variable *i* (line number 62), then this formula evaluates to 1. Otherwise, if the current code fragment is the increment of *i* (line number 64), then this formula computes the value of cell F9 plus 1. If neither of these apply, then the value of this formula equals the value of cell F9.

The formula entries are universal in the sense that they are identical throughout the column except that the row numbers referring to the current and the previous rows are adjusted.

The formula in the first column (labeled with "no") determines the line number of the current step based on the line number of the previous step and the

value of the condition in the previous step. For example, the formula entry in cell A3 of the sheet shown in Figure 4 is as follows.

```
=IF(A2=0,30,IF(A2=30,31,IF(A2=31,50,
IF(A2=50,51,IF(A2=51,61,IF(A2=61,62,
IF(A2=62,63,IF(AND(A2=63,C2=TRUE),70,
IF(AND(A2=70,C2=TRUE),80,IF(AND(A2=70,
C2=FALSE),64,IF(A2=80,64,IF(A2=64,63,
IF(AND(A2=63,C2=FALSE),90,90))))))))))))
```

This formula compares the line number of the previous step to all line numbers in the program and, possibly depending on the value of condition in the previous step, decides the current line number.

The formula for the second column (labeled as "code") is straightforward. It compares the current line number with all line numbers and, when the numbers match, returns the string for the code fragment corresponding to the number.

The formula for the third column representing the value of the condition, when the current code fragment is a condition expression, is similar to the one for memory cells.

4.2 How to Generate Formula Entries

In this section, we show how the formula entries of a tracesheet is generated from a source code of Java program.

First, we parse the source code and get an Abstract Syntax Tree (AST) using Eclipse JDT libraries. Then we analyze the AST and create a *transition diagram*, which is a directed graph where the vertices are code fragments and the edges are transitions. A transition is *unconditional* if it represents the usual sequential flow in the execution. It is a *true-transition* it corresponds to conditional branches prescribed by an if-statement or a for-statement and occurs when the condition evaluates to true. *False-transitions* are defined similarly. In addition to the transition graph, we compute the list of *updates* for each variable. An update for a variable is a pair consisting of the line number of a code fragment that is an assignment to the variable and the right-hand-side expression of the code fragment.

Given the transition graph and the list of updates, it is rather straightforward to generate formula entries of the tracesheet. We omit the details.

5 CONCLUDING REMARKS

We have proposed the use of tracesheets in introductory programming courses, discussed the motivation and potential benefits of their use, and described a preliminary implementation of a tracesheet generator for Java. We plan to evaluate the effectiveness of tracesheets through their use in actual classes and sessions and in a web-based programming course we are developing.

REFERENCES

Baecker, R. (1998). Sorting out sorting: a case study of software visualization for teaching computer science. In *SoftwareVisualization: Programming as a Multimedia Experience*, pages 369–381. MIT Press.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290.

Lahtinen, S.-P., Sutinen, E., and Tarhio, J. (1998). Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing*, 9(3):337–349.

Levy, R. B.-B., Ben-Ari, M., and Uronen, P. A. (2003). The Jeliot 2000 program animation system. *Computers and Education*, 40(1):1–15.

Microsoft Corporation (1985). Microsoft excel. http://office.microsoft.com/en-us/excel.

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 373–376. ACM Press.

Sajaniemi, J. and Kuittinen, M. (2003). Program animation based on the roles of variables. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 7–16. ACM Press.

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., and Edwards, S. H. (2010). Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3):1–22.

Sun Microsystems (1995). Java. http://www.oracle.com/technetwork/java/index.html.

Sutinen, E., Tarhio, J., and Teräsvirta, T. (2003). Easy Algorithm Animation on the Web. *Multimedia Tools and Applications*, 19(2):179–194.

Timsit, C. and Zertal, S. (2010). Using spreadsheets to teach computer architecture. In *CSEDU 2010 2nd International Conference on Computer Supported Education*, pages 101–105. INSTICC.

Urquiza-Fuentes, J. and Ángel Velázquez-Iturbide, J. (2009). A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems. *ACM Transactions on Computing Education*, 9(2):1–21.