Modeling and Automatic Code Generation Tool for Teaching Concurrent and Parallel Programming by Finite State Processes

Edwin Monteiro[0000-0002-9623-3233], Kelvinn Pereira[0000-0003-2900-0728], and Raimundo Barreto[0000-0001-8494-4225]

Federal University of Amazonas, Institute of Computing, Brazil
{edwin,kdsnp,rbarreto}@icomp.ufam.edu.br

Abstract. Understanding concurrent and parallel programming can be a very hard task on first contact by students. This paper describes the development and experimental results of the FSP2JAVA tool. The proposed method starts from concurrent systems modeling through Finite State Processes (FSP). After that, the method includes an automatic code generation from the model. This goal is achieved by a domain-specific language compiler which translates from the FSP model to Java code. The FSP2JAVA tool is available for free download in the github site. We argue that this tool helps in teaching concurrent systems, since it abstracts all complex languages concern and encourages the student to be focused at the fundamental concepts of modeling and analysis.

Keywords: FSP · concurrent programming · code generation · teaching.

1 Introduction

Concurrent programming is a paradigm used in building programs that make use of the simultaneous execution of multiple tasks that can be implemented as separate programs or as a single program that triggers multiple threads in parallel. The main advantage of using concurrent programming is the increased performance since it is possible to increase the number of tasks performed over a given period of time. The major challenge of concurrent programming is resource sharing, communication, and interaction between programs that run concurrently. The reason for this challenge is that parts of a program can now execute in an unpredictable order. Therefore, errors can occur depending on the order of execution of each task. However, usually such errors are difficult to find.

There are several examples of concurrent issues reported in the literature. One of them is Therac–25, which caused massive radiation overdoses. Another example was the case of Knight Capital Group, which lost $460 million in 45 minutes as presented in Kirilenko et al. [6]. In both cases, the systems programming had few revisions and there were no checks to certify that the software had been developed correctly. Taking care of developing concurrent systems should be present from the initial training of professionals. However, teaching this paradigm

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

is a major challenge in undergraduate classes since students have a hard time understanding the theory behind a problem. They usually do not take the time to think of effective solutions to problems. Instead, usually they go directly to programming attempting to get a possibly correct solution, which may lead to undetected programming errors. The problem of taking it a step further may be justified by the fact that the student deals with a complex level of programming never seen before and unrelated to the problem itself. According to [3], this may be mainly affected by the following factors: (i) a new mindset required by programming multithread; (ii) the behavior of a multithreaded program is dynamic, which makes the debugging task very difficult; and (iii) synchronization is more difficult than expected.

This paper presents FSP2JAVA, a tool for modeling concurrent systems through Finite State Processes (FSP) introduced in [8]. The goal is to prevent students to be in contact with the coding at an early time. Thus, the main aim is to facilitate the understanding of the fundamental concepts of concurrent programming in order to make the coding step easier. For methodological purposes, the high level code is shown only at the end of the modeling stage.

This paper is organized as follows. Section 2 reviews related work. In Section 3 we introduce the concepts of Finite State Process. Sections 4 and 5 detail the FSP2JAVA tool. Section 6 presents the planning and execution of experiments. Finally, Section 7 discusses the final considerations and future work.

2 Related Works

Learning the concepts of concurrent programming is essential for computer science students. The most common method of concurrent programming is one that adopts multithreaded programming. However, changing it from sequential paradigm causes significant problems for students, as concurrent programming interfaces are often more complex than necessary, causing students to spend time learning system details rather than the fundamentals as reported in [3].

Among the modeling tools found for teaching distributed systems, we can mention: (a) SPIN, (b) SCML, (c) LTSA and (d) FSP2JAVA. SPIN is an efficient verification system for distributed software system models. It was used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code to control switchboards [4]. SCML is a tool that can be used to simulate a system of concurrent processes that communicate through shared variables. Mechanisms for defining non-determinism, atomic actions, and process synchronization are supported. In addition, SMCL includes a prototype for verifying basic security properties, such as mutual exclusion and deadlocks, using the model check technique [2]. The LTSA tool [7] is used at numerous universities around the world along with the book by Magee and Kramer [8]. The LTSA tool compiles the FSP specifications on a state machine and resembles the non-deterministic finite automaton. LTSA also features viewing and animating labeled transitions through graphical interfaces. Both FSP and LTSA are widely used.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

The proposed tool is called FSP2JAVA. It was developed by [11] and improved by [12]. This tool receives a model in FSP and checks if the modeling is properly specified. If the model is correct, the second step is automatic code generation which consists of transforming the FSP language into Java language.

3 The Finite State Process (FSP)

The Finite State Process [9] is a formal language based on process algebra [1] to model the behavior of concurrent systems. The FSP notation is derived from the process algebra CSP and can represent a system by primitive processes (a single thread) or a composition of processes (multithreaded).

3.1 Primitive Process

The primitive process characterizes the execution of a sequential program. The term primitive is related to the basic structures of a programming language, such as the choice, guard condition, recursion, and alphabet extension. The main operations are described below:

Action Prefix `(a->P)`: Describes a process that performs the *a* action, and then behaves exactly as specified in the `P` process. Practically, the action prefix defines a transition between states.

Choice `(a->P | b->Q)`: Process behavior is defined by `a` or `b`. After an action is performed, subsequent behavior is described by `P` if the first event was `a`, or by `Q` if the first event was `b`.

Process STOP: Sometimes it is necessary to finish the execution of a process. When STOP is called, no further action is evaluated.

Alphabet Extension +{`a,b,...,z`}: A process can behave only through the actions contained in its alphabet, although the opposite is not valid. In certain situations the alphabet may be extended with actions not previously defined in the modeling. This inclusion is very common to prevent another process from performing a particular action.

Guarded Action `(when B a->P)`: Actions of type are eligible only when Boolean condition `B` is satisfied, otherwise `a` is not a valid action.

Indexing `P = (input[i:0..9]->output[i]->P)`. Allows the writing of processes and actions that assume multiple finite values in a simplified manner.

3.2 Composite Processes

Processes are concurrent and can perform actions in parallel. These actions may or may not be shared.

Parallel Composition `(P || Q)`: Expresses the parallel execution of processes `P` and `Q`. Thus, actions are merged as they are executed. If two or more actions are shared, then the processes that contain them are synchronized. The `||` operator is the parallel composition operator.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

Process Instances (a:SWITCH || b:SWITCH): Operation applied to differentiate distinct instances (in this case a and b) from the same process (in this case, SWITCH).

Set Labeling $\{a_1, ..., a_n\}$::P Add a prefix to all actions belonging to the alphabet of P. If {x,y,z} belongs to the alphabet, then the operation results in $\{a_1.x, a_2.y, a_3.z\}$.

Re-labeling $/\{new_1/old_1, ..., new_n/old_n\}$: This operation is applied to ensure that composition of processes synchronize specific actions. Although common in composite processes, the operation can be applied to primitive processes.

4 FSP to Java Method

This paper extends the works presented in [8] and [11]. Therefore, the goal remains to evaluate an FSP model, interpret its behavior, and ultimately generate Java code from the transformation rules described later. In addition, this research increases the FSP instruction set accepted by the tool and facilitates interaction with users, as the entire process that starts at the modeling stage and ends at the code generation stage is focused exclusively on the tool.

4.1 Translation

Consider the model described below in FSP:

```
COIN=(toss->HEADS|toss->TAILS),
HEADS=(heads->COIN),
TAILS=(tails->COIN).
```

It models the tossing of a coin that assumes two states, heads or tails. Note that the choice occurs non-deterministically, since the same action implies distinct processes. In FSP models, according to the specification proposed in [8], the processes are written in uppercase and actions in lowercase. The simple distinction in writing allows us to establish the basic rule of language transformation: processes and actions are transformed into classes and methods, respectively. The following example illustrates applying this rule to generate the Java code corresponding to the COIN model:

```
public class COIN{
    public void toss_0(){
        System.out.println("toss");
    }
    public void heads_0(){
        System.out.println("heads");
    }
    public void tails_0(){
        System.out.println("tails");
    }}
```

Since they are syntactically distinct languages, some adaptations are required during the transformation. For instance, FSP algebra allows an action to be

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

duplicated in the same process or in a new process. Thus, transforming an action into a method adds an identifier to the method name. This is done to allow the distinction between actions of the same name because in Java there are no duplicate methods that have exactly the same behavior. To represent the state transition, that is, when an action is reached in FSP, the respective methods contain the `Println` function that prints the name of the action associated with the method. This was the strategy adopted to simulate the sequence of actions FSP achieved during the execution of the respective transformed code. Another adaptation consists of local processes, for example, HEADS and TAILS are not transformed into classes. However, their methods are incorporated into the COIN class that corresponds to the main process. This identifier strategy is adopted in actions with the same name in different processes, this includes the actions of local processes, since everything will be put together to the same class.

4.2 Transformation Rules

Primitive Processes: These are transformed into classes that implement the Runnable interface. Thus, each class must contain an attribute of the class Thread. This is justified because Java does not contain multiple inheritance, so the class Thread is used without the need for inheritance. In addition to implementing the Runnable interface, the process class must contain the implementation of the `run` method that allows concurrent execution of the processes. The following example illustrates the transformation rule of primitive processes:

FSP: `P = (a -> P).`

Java:

```
public class P implements Runnable{
    Thread threadP;
    P(){threadP = new Thread(this);
        threadP.start();
    }
    public void a_0(){
        System.out.println("a");
    }
    public void run(){
        try{
            while(true){
                Thread.sleep(1000);
                a_0();
            }
        }catch(Exception e){}
    }}
```

STOP: It is analogous to the process described above with the difference that there is no infinite loop. If the STOP process is selected during model interpretation, then the Java equivalent code is represented by an interrupt method call after the last action `a` in this example. Then two statements are inserted before the end of the run method:

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

```
System.out.println("STOP");
threadP.interrupt();
```

Indexed Processes: In the case of indexed processes, the name of each method of the class, in addition to containing the action name FSP, also contains the unique action identifier and index of the respective indexed process. The following example illustrates the Indexed Processes transformation rule:

FSP:

```
const N = 1
P = P[0],
P[i:0..N] = ( a -> P[0] | b -> P[1]).
```

Java:

```
public class P implements Runnable{
    public void a_0_0(){
        System.out.println("a");
    }
    public void b_0_0(){
        System.out.println("b");
    }}
```

`P[i:0..N]` is equivalent to writing N distinct `P` processes that communicate with each other from recursive calls determined by the process index that is triggered by an action. This index acts as the process selector. In the actions, the first digit represents the action identifier (this allows differentiating actions with the same name in a modeling) as explained earlier. The second digit characterizes which process the action belongs to. For example, the `P` process starts with the index 0, so if the first action is `b`, then the action will be called `b_0_0`. From `b_0_0` process `P` receives the index 1. If `a` is selected then the action name is in the form `a_1_1` because this action `a` is associated with `P[1]`. Note that for each indexed process, there is a constant amount of associated actions (in this case there are two).

Indexed Actions: The methods created from indexed actions, in addition to printing the action name, also print their index in square brackets.

FSP:

```
const N = 2
BUFF = (in[i:0..N]->out[i]->BUFF).
```

This type of indexing allows the creation of a large number of distinct actions in a few rows. Each action is named together with a value of variable `i` in the range `0..N`. The code generation is similar to that presented for indexed processes, but what changes is the print of each method. For instance, consider the following code: `System.out.println("in[0]")`. In this case, there is an action that takes the form `in[0]` or `out[0]` for each indexed action such that 0 is any value in the `i..N` range. Unlike the previous indexing example, there is a single process and N actions so that the ith action `in[i]` is succeeded by the ith action `out[i]`. Thus, the run method will always have an `in` action followed by an `out` action of the same index.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

Parallel Composition: A new class is created with the same name as the parallel composition model. This class instantiates and executes all primitive processes previously modeled in parallel composition. Consider a practical example of the roller coaster problem where the car's Mth passenger is controlled by a turnstile that allows exactly 3 passengers at a time. As long as there is room in the car, passenger boarding is cleared. When M = 3, the car starts rolling down and up, and a new M-capable car arrives to board new passengers. The modeling below represents the problem described in FSP:

```
const M = 3
TURNSTILE = (passenger -> TURNSTILE).
CONTROL = CONTROL[0],
CONTROL[i:0..M] = (when(i<M) passenger->CONTROL[i+1]
                  | when(i==M) depart->CONTROL[0]
                  ).
CAR = (depart->CAR).
||ROLLERCOASTER = (TURNSTILE || CONTROL || CAR).
```

In Java the ROLLERCOASTER process model is adapted to the following class:

```
public class ROLLERCOASTER{
   public static void main(String args[]){
      Monitor passenger_shared = new Monitor(4);
      Monitor depart_shared = new Monitor(2);
      TURNSTILE obj_turnstile = new
      TURNSTILE(passenger_shared);
      CONTROL obj_control = new
      CONTROL(passenger_shared, depart_shared);
      CAR obj_car = new CAR(depart_shared);
   }
}
```

The ROLLERCOASTER is the main class, because it is from it that the communication between the processes begins. Due to passenger dispute over access to the car, only three passengers are allowed to access the car at a time. To manage the seats available in the car, two monitors are created, one for passenger control and one for starting the car (releasing the car to new passengers). This release simulates the availability of a new car. The other processes, TURNSTILE, CONTROL, and CAR are instantiated in this main class. For each instance is passed a parameter of type Monitor corresponding to each process. For example, the turnstile deals with passengers. Therefore, only the passenger_shared monitor is passed as a parameter.

The TURNSTILE process is defined in terms of the following class:

```
public class TURNSTILE implements Runnable{
   Thread threadTURNSTILE;
   Monitor passenger_shared;
   TURNSTILE(Monitor passenger_shared){
     this.passenger_shared = passenger_shared;
```

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

```
      threadTURNSTILE = new Thread(this);
      threadTURNSTILE.start();
   }
   public synchronized void passenger_0() throws
       InterruptedException{
      passenger_shared.dec();
      if(passenger_shared.inc()){
         System.out.println("passenger");
      }
   }
   public void run(){
      try{
         while(true){
            Thread.sleep(1000);
            passenger_0();
            Thread.sleep(1000);
            passenger_0();
            Thread.sleep(1000);
            passenger_0();
            Thread.sleep(1000);
         }
      }catch(Exception e){}
   }}
```

As defined in FSP, this process occurs in parallel with CONTROL and CAR, so it implements the Runnable interface to run concurrently. A thread type object is defined to concurrently execute with other model actions. In addition, a monitor type attribute is set to receive from the monitor that was passed by parameter through the class constructor (as shown in ROLLERCOASTER).

The passenger action described in the above model is defined in passenger_0 method. As shown, this method uses the keyword synchronized in order to ensure that this method is executed by only one thread at a time. The passenger_shared.inc() condition assumes two values: True or False. If True, the car contains the maximum number of passengers, so all passengers accessed the feature. If false, there is still room for more passengers in the car. For each passenger who has gained access to the car, the method prints the name "passenger" to simulate the behavior of the FSP model. Finally the run method is responsible for executing the request of M passengers. Like most FSP models, this model works continuously and never leaves the loop. The number of times a method is called on the loop run is related to the depart of a car and the arrival of a new one. For this example only one car departed, so there are 3 calls to the passenger method.

The transformation rule to the CAR and CONTROL processes is analogous to the adaptation made for TURNSTILE. It is worth noting that CONTROL will have two methods, as its constructor receives two monitors for passenger and depart actions as presented above.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

5 The FSP2JAVA Tool

The FSP2JAVA (Fig. 1) is a tool designed to model real systems through the FSP language that abstracts the complexity and effort to master high level languages during the transition from sequential to concurrent paradigm. Note that the tool is available for use in the GitHub repository [12]. Based on feedback obtained from users using the previous version of the [11] tool, the current version of FSP2JAVA has a integrated user-friendly graphical interface that allows you to model, to analyse and to generate a concurrent Java code. This version also has the user manual, warning screen and error messages, interactive model interpreter and automatic code generation menu. All of these features will be described below.

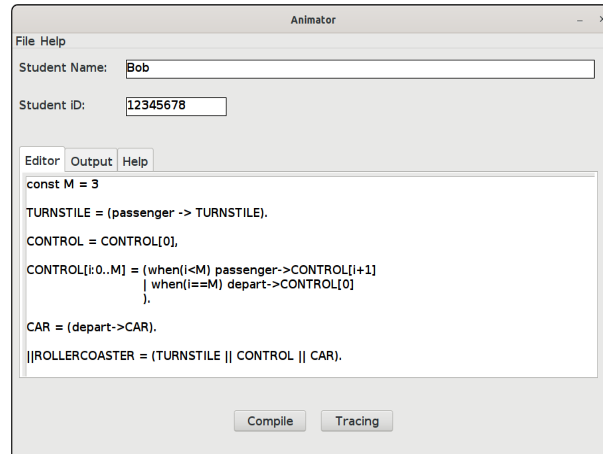

Fig. 1. FSP Modeling Screen.

5.1 FSP2JAVA Components

The FSP2JAVA tool contains two windows: Animator and Trace. The first of these contains the components below.

File: This menu allows to create, save or edit an FSP model. There are three options available in the File menu:

- **New**: Clears all content present in the modeling area.
- **Save**: Saves current edit area to a file with the extension ".fsp".
- **Open**: Opens a new text file.

Help: This menu provides a tool user manual in a new tab next to Output.

Editor tab: This tab is a text editing area, in which the user writes codes in FSP that are later converted to code in Java.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

Output tab: This tab (see Fig. 2) serves to alert if the compilation of the code was successful or not. If syntax errors occur, the error location indicating the row, column and the error itself are shown.

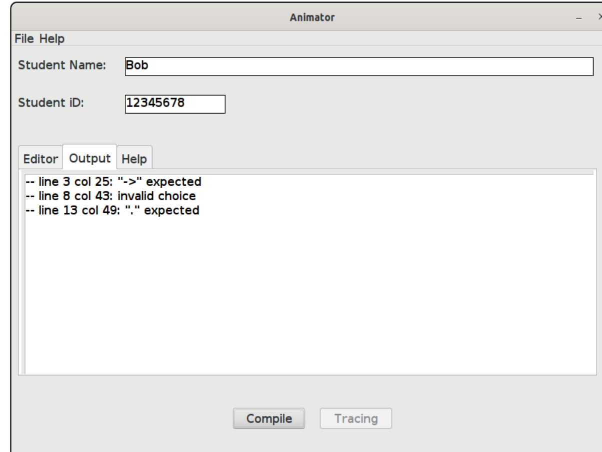

Fig. 2. The ROLLERCOASTER example containing a syntax error

Compile button: Allows the compilation of FSP code present in the editing area of the Editor tab. When compilation succeeds, the Tracing button is enabled.

Tracing button: The Tracing button is only available when compiling a model has been successful. In this case, the button opens the Trace window where the interactively selected actions are later added to the generated code. The tool always generates code with valid execution traces.

The components in the Trace window (see Fig. 3) are described below:

Trace History: A text area that shows all the traces produced so far.

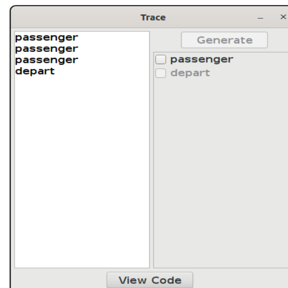

Fig. 3. The Trace window with the successfully compiled ROLLERCOASTER code.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

Checkboxes: The checkboxes correspond to the actions coming from the FSP code that was compiled. Whenever an action is eligible, the box allows its selection.

Generate button: It allows the generation of Java code according to the actions selected so far.

View Code button: It opens all Java code generated in the system's default text editor according to the actions selected in the checkbox. The View Code button is available only when the Generate button is clicked.

6 Experiment Planning and Application

The experiment was carried out with two undergraduate students from Software Engineering and Computer Science courses at the Federal University of Amazonas (UFAM). Although few students, they presented different profiles. One student already known about concurrent programming, while the other had no knowledge of the subject. This feature was essential, since the collection of opinions at the end of the experiment could contribute significantly to the improvement of the tool. The purpose of the experiment was to evaluate the contribution of FSP2JAVA as a mediator tool for concurrent programming teaching. In order to be able to perform the experiment, a class was taught on the topic of concurrent systems modeling by finite state processes. During the explanation of the concepts, some exercises were applied and solved in the FSP2JAVA tool with the intention of making the students more comfortable with the tool.

The FSP2JAVA experimentation took place in two steps as follows:

a) Modeling and analysis of the tool. For this purpose, the classic roller coaster problem was proposed as an exercise. In this problem, passengers arriving at the boarding platform must be registered at the roller coaster controller by a turnstile. Thus, the controller allows the car to depart only when there are enough passengers on the platform so that the car is occupied until its maximum passenger capacity M is reached. This problem was chosen because it adopts multiple threads with synchronization. Thus, all the features of the tool could be evaluated. To solve the problem, students were given one hour to use the FSP2JAVA tool without consulting the other candidate or instructor. After modeling, the students executed and evaluated the code generated by the tool.

b) Filling questionnaire - Two questionnaires were applied to gather students' perception of the tool. The first was related to the generated model, and the second to Java-generated code by FSP2JAVA. Both were applied at the end of the use of the tool. In this context, the model proposed by [10] was adapted to evaluate the motivation of the use of modeling tools based on the following aspects: intention to use, ease of use, correctness, reliability, satisfaction and usefulness. Tables 1 and 2 present the aspects evaluated by the participants, where they should report their degree of agreement by choosing the options presented in a 6-point Likert scale [5]: strongly disagree, widely disagree, partially disagree, neutral, partially agree and totally agree.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

Table 1. Items taken into consideration when evaluating modeling.

Item	Correctness	Reliability	Facility	Quality	Satisfaction	Utility
Description	When I use the support tool, it works correctly to model a distributed system.	I trust the validation of the model by the support tool.	The support tool is easy to use to model a distributed system.	The support tool is useful for modeling distributed quality systems.	I am satisfied with the validation of the model by the support tool.	I would use the support tool when I wanted to model quality distributed systems.

Table 2. Items taken into account to evaluate code generation.

Item	Correctness	Reliability	Facility	Quality	Satisfaction	Utility
Description	When I use the support tool, it works correctly to generate code for distributed system.	I trust the code generated by the support tool.	The support tool is easy to use to generate code for a distributed system.	The support tool is useful for generating code for quality distributed system.	I am satisfied with the code generated by the support tool.	I would use the support tool when I wanted to generate code for a distributed system.

In addition to these questionnaires, some open questions were applied for the collection of perception, possible difficulties, problem detection and suggestions for improvement: (a) "What is your opinion on the tool's feedback to support your learning/performance when modeling concurrent systems?"; (b) "What made it difficult to use the supporting tool to create Finite State Process based models?"; (c) "What would you change about using the support tool to improve your learning/performance when modeling concurrent systems?"; (d) "What is your opinion on the feedback of the tool to support your learning/performance when generating code from concurrent systems?"; (e) "What made it difficult to use the backup tool to generate Java source code from concurrent systems?"; and (f) "What would you change about using the support tool to improve your learning/performance when generating code from concurrent systems?".

The results obtained from the experiment indicate that the tool has great potential to be adopted in the teaching of concurrent programming. The data obtained, see Fig. 4, on the use of the tool for teaching concurrent system modeling point to a broad agreement on all items in the Table 1. Students believe in the correct functioning of the tool and are satisfied with validating their models so that they would use it again to model concurrent systems. With respect to code generation, Fig. 5 allows us to understand that there is complete agreement on all items questioned in the Table 2.

From the experiments results, it can be noted that the tool worked correctly for the designated purpose so that the students were safe and confident about its use. The analysis also allows us to conclude that the generated code behaves as expected, because the result generates satisfaction to the students regarding the use of the tool in critical situations where it is necessary to generate quality concurrent code.

In order to understand which aspects had a positive or negative impact on the students' evaluation regarding the use of FSP2JAVA, the open questions

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

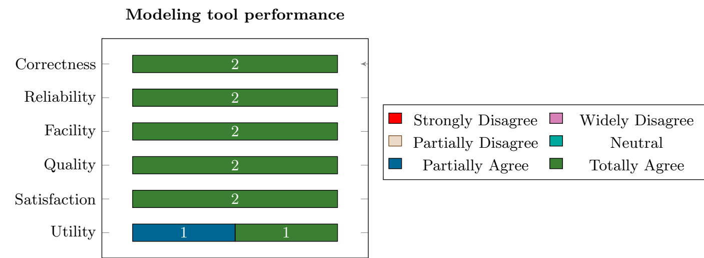

Fig. 4. Student evaluation considering the modeling aspects of FSP2JAVA.

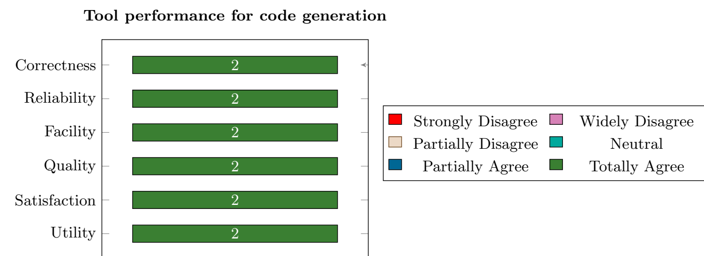

Fig. 5. Evaluation results considering the FSP2JAVA automatic code generation.

in the questionnaire were analyzed to obtain feedback and identify facts that justified the degree of agreement on the items evaluated.

The following are some positive and negative comments, where Pi corresponds to the i-th participant:
"Easy to remember, to identify things (buttons, desktop, features in general), it has a friendly interface" – Positive (P02).
"I would add some examples of FSP in the user manual" – Negative (P02).
"I would improve feedback on syntax errors generated or suggested by the system, thus helping the user to determine syntax errors faster." – Negative (P01).

7 Final Considerations

This paper introduced FSP2JAVA, an automatic modeling and code generation tool for concurrent systems. The purpose of this research was to facilitate the teaching of programming for early classes of this paradigm from a tool that abstracts all the concern with complex languages and encourages the student to master the fundamental concepts of this subject. The information obtained through the questionnaires and open questions corroborated the effectiveness of the tool in teaching concurrent programming. However, the responses collected indicate that feedback needs to be improved, as reported, so that students understand modeling errors more intuitively to allow for a growing learning curve.

Due to the fact that FSP2JAVA experimentation was performed with few students, the resuWlts obtained cannot be generalized. Therefore, in future works,

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44

it is intended: (i) to improve tool feedback; (ii) to add examples of FSP modeling and transformation rules to the manual; (iii) to apply new experiments with more students with different levels of knowledge; (iv) to observe the impact of using the tool depending on the lesson plan covered; (v) to add code generation in other programming languages such as C/C++ and Python; (vi) to make the tool accessible on the web; and (vii) to propose metrics for code quality analysis for distributed systems.

Therefore, it can be concluded that FSP2JAVA has great potential to be a mediator between students and teachers interested in learning and teaching concurrent programming in an effectively way.

Acknowledgments. This research, in accordance with Article 48 of Decree nº 6.008/2006, was funded by Samsung Electronics of Amazônia Ltda, under the terms of Federal Law nº 8.387/1991, through agreement nº 003, signed with ICOMP/UFAM.

References

1. Baeten, J., van Beek, D.A., Rooda, J.: Process algebra. Handbook of Dynamic System Modeling pp. 19–1 (2007)
2. Ben-Ari, M.: Teaching concurrency and nondeterminism with spin. In: ACM SIGCSE Bulletin. vol. 39, pp. 363–364. ACM (2007)
3. Carr, S., Mayo, J., Shene, C.K.: Threadmentor: a pedagogical tool for multithreaded programming. Journal on Educ. Resources in Computing **3**(1) (2003)
4. Holzmann, G.J.: The SPIN model checker: Primer and reference manual, vol. 1003. Addison-Wesley Reading (2004)
5. Jamieson, S., et al.: Likert scales: how to (ab) use them. Medical education **38**(12), 1217–1218 (2004)
6. Kirilenko, A.A., Lo, A.W.: Moore's law versus murphy's law: Algorithmic trading and its discontents. Journal of Economic Perspectives **27**(2), 51–72 (May 2013). https://doi.org/10.1257/jep.27.2.51
7. Lang, F., Salaün, G., Hérilier, R., Kramer, J., Magee, J.: Translating fsp into lotos and networks of automata. Formal Aspects of Computing **22**(6), 681–711 (2010)
8. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. Wiley Publishing, 2nd edn. (2006)
9. Magee, J., Kramer, J., Giannakopoulou, D.: Analysing the behaviour of distributed software architectures: a case study. In: Proc. 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems. pp. 240–245 (1997)
10. Martínez-Torres, M.R., Toral Marín, S., Garcia, F.B., Vazquez, S.G., Oliva, M.A., Torres, T.: A technological acceptance of e-learning tools used in practical and laboratory teaching, according to the european higher education area. Behaviour & Information Technology **27**(6), 495–505 (2008)
11. Monteiro, E., Rivero, L., Barreto, R.: Uma ferramenta de suporte ao ensino de modelagem de sistemas distribuídos críticos: Uma experiência prática (in portuguese). In: Brazilian Symposium on Computers in Education. vol. 29 (2018)
12. Nunes, K., Monteiro, E., Barreto, R.: FSPTOJAVA: A modeling and automatic code generation tool. https://github.com/kelvinnpereira/pibic-fsp (2019)

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_44