Publication I

Jan Lönnberg and Anders Berglund. Students' understandings of concurrent programming. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, pp 77–86, Koli, Finland, April 2008.

Students' understandings of concurrent programming

Jan Lönnberg[1] **Anders Berglund**[2,1*]

[1] Department of Computer Science and Engineering
Helsinki University of Technology,
Espoo, Finland,
Email: jlonnber@cs.hut.fi

[2] Department of Information Technology
Uppsala Computing Education Research Group, UpCERG
Uppsala University,
Uppsala, Sweden
Email: anders.berglund@it.uu.se

[*] Temporary affiliation

Abstract

This paper describes a qualitative, explorative study of how students understand some concepts in concurrent programming. The study is based on interviews with students regarding the final programming assignment in a concurrent programming course. We use phenomenography to analyse the students' statements about tuple spaces, the concurrent data structures on which the assignments are based, and to find the different ways in which they understand writing and debugging a concurrent program. We then discuss the effects of these understandings on how students construct concurrent programs, how teaching can be improved to form more useful understandings and how software tools can be designed to support the development of concurrent programs.

1 Introduction

Concurrent programming is both an important technique and a challenge. On the one hand, concurrent programming provides a way to make effective use of parallel and distributed systems and structure systems that perform many simultaneous tasks. On the other, the unpredictability of interaction between concurrently executing processes also introduces many pitfalls in the software development process that may result in software defects that are hard to find.

The research presented in this paper is part of a larger project with the long-range goal of making it easier to produce correct concurrent programs (i.e. programs that consistently produce the right results despite the aforementioned unpredictability) by supporting the detection and elimination of defects (or *bugs*).

In order to effectively develop methods and software tools to help find and eliminate bugs in concurrent programs, it is necessary to understand the bugs that can appear in these programs and the programmers' reasoning underlying these bugs. While insights about bugs per se can be gained by examining code, an understanding of programmers' reasoning, which is the focus of this study, must be based on empirical, explorative work.

In Section 2, we describe our long-range research goals and how they relate to this study. In Section 3, we explain the relevant concepts of concurrent programming and previous research on students' understanding of concurrent programming. In Section 4, we summarise the basic concepts of phenomenography and how it has been used in computer science education research.

In Section 5, we describe how we interviewed the students about a concurrent programming assignment and used phenomenography to distill the students' different understandings from the interview transcripts. In Section 6, we present these understandings and then, in Sections 7 and 8, we discuss how these types of understanding affect the students and what they mean for teaching concurrent programming and for developing ways to support students in developing correct concurrent programs.

2 Research questions

Our long-range goal is to help programmers create software that works correctly by aiding them in understanding, testing and debugging concurrent programs. We approach this by developing methods and tools to help programmers understand what a concurrent program does, especially when it is not working as expected. We intend to achieve this through visualisation of program execution and data.

Price et al. (1993) state that while software visualisation (SV) "has tremendous potential to aid in the understanding of concurrent programs", few SV systems have seen production use, especially in the domain of tools for professional programmers. They note that when a SV system is designed, the content to be shown must be selected according to the goals of the system, which, in turn, are based on the requirements of the users.

Based on the above, our large-scale approach is to first identify the needs of the programmers and then design solutions to address them. The general questions we therefore seek answers to are:

- What defects do programmers introduce in concurrent programs, and why?
- Which of these defects are difficult to locate and why?
- What sort of visualisations (of a program execution or model checker counterexample) can assist a programmer in finding these most problematic defects, and how well do they work?

The intent of this study is to complement research into defects found in concurrent programs and visualisation of concurrent programs with information on different ways of thinking about concurrent programming. This will help when trying to reason about how a defect was introduced and when constructing visualisations to support programmers in understanding what their program does, especially when it does not behave as expected.

Naturally, this study can also be seen as part of the process of improving the teaching of concurrent programming by examining how students understand the concurrent programming concepts they have been taught.

2.1 Aims of this study

The purpose of the current work is to shed light on how programmers understand some core concepts, so that these insights can serve as a platform for exploring possible sources of errors, especially those that stem from insufficient or incorrect understandings. A better understanding of errors will help us make better debugging and development software.

We have chosen to focus on studying the understandings of students for three different reasons. Firstly, we assume that students, particularly master's and doctoral students, can be used as a source of information for designing tools and approaches that will also be useful for professional programmers. Secondly, we can collect and analyse large amounts of data from students, with less effort than from commercial software developers. Thirdly, an understanding of how students understand and approach concurrent programming is also important for improving how concurrent programming is taught. We briefly explain the relevant aspects of concurrent programming and previous research on how students understand it in Section 3.

In this study, we explore the different ways in which students in a concurrent programming course understand a concurrent data structure, the *tuple space*, as well as their various understandings of what developing and debugging a concurrent program means. We do this using an empirical, qualitative research approach called *phenomenography* (Marton & Booth 1997), which we will describe in Section 4.

3 Background

In this section we will briefly introduce concurrent programming as it pertains to this study and is perceived by experts (the textbook perspective, in Subsection 3.1) and students (the results of research into students' understanding of concurrent programming, in Subsection 3.2).

3.1 Concurrent programming

A *concurrent* program is a program that contains two or more *processes* that co-operate to achieve a goal, where each process is a set of sequentially executed instructions like a sequential program. These processes may actually be executing on separate hardware (multiple processors or execution cores in a single computers, or geographically separated computers). Alternatively, the concurrency may be simulated on a single processor by executing one process at a time for a brief period of time (Andrews 2000, Ben-Ari 2006).

Most workstation and server operating systems (e.g. Unix) provide support for dividing running processes between processors. Alternatively, a simulator can be used to simulate a multiprocessor machine for research or teaching purposes (Pears 1995).

Although a primer on concurrent programming is beyond the scope of this article, we will briefly describe the aspects of concurrent programming that are relevant to this study.

The most important distinction between sequential and concurrent programs is the inherently nondeterministic behaviour of concurrent execution; it is unknown how much of one process is executed during the time another one executes an instruction. The greatest challenge in writing concurrent programs is getting concurrent processes to reliably interact properly in the face of this nondeterminism (Andrews 2000, Ben-Ari 2006).

There are several approaches to ensuring correctness despite nondeterminism, including *deductive proofs* (usually manually constructed) and *model checking*, which can be based on a simplified computational model, such as in Spin (Holzmann 1997), or an actual programming language, e.g. Java PathFinder (Visser et al. 2003).

In order to co-operate, processes must be capable of communicating with each other. Many different mechanisms for interprocess communication (IPC) are available for this; only those relevant to this study will be described here.

One of the most common IPC mechanisms is *shared memory*: memory that can be written to by many different processes. This is typical of multithreaded systems written in languages such as Java.

Other IPC mechanisms are typically provided to complement shared memory, some of which focus on preventing processes from proceeding with potentially harmful execution. The simplest is the *lock* or *mutex*, which allows the programmer to designate *critical sections* in a program, only one of which can run at a time. The *semaphore* can be considered an extension of the lock; it is essentially a shared non-negative integer value accessible through two operations: `V`, which increases the value of the semaphore by one, and `P`, which waits until the value of the semaphore is positive and then decreases it by one. A variety of *message-passing* mechanisms are often used in distributed systems; as the name implies, these involve processes sending messages to each other and receiving them either by waiting for a message or by buffering messages for later reading.

Gelernter (1985) describes, as a central part of the distributed programming language *Linda*, an interprocess communication mechanism called a *tuple space*. As its name implies, a *tuple space* consists of a space containing *tuples*, data records consisting of a *tag* (an identifier for the type of tuple) and (an ordered list of) zero or more data values. A tuple space can be accessed through three operations: `in()`, `out()` and `read()`. `out()` takes a tuple as an argument and inserts it in the tuple space. `in()` takes as its argument a pattern consisting of a tag and zero or more data values or formal parameters. As soon as a matching tuple is found, `in()` fills all the formal parameters with the corresponding values from the matching tuple, removes the tuple from the space and returns. `read()` behaves like `in()`, but does not remove the tuple from the space.

Tuple spaces can be used in many different ways. Therefore, they can be considered to be generalisations of several different IPC mechanisms. On the one hand, they are a form of shared memory; on the other, they can be seen as semaphores with associated data or a message-passing mechanism; all of these can easily be implemented using a tuple space (Gelernter 1985).

3.2 Students' understanding of concurrent programming

Ben-Ari & Ben-David Kolikant (1999) describe how high-school students' concurrent programming conceptions and working methods change during a course on the subject, focusing on difficulties faced by the students in dealing with concurrent programming. They found that students have problems with limiting themselves to operations permitted by the concurrency model, make assumptions based on informal concepts rather than use formal rules and avoid using concurrency. The students also applied development approaches that work well with sequential programming but not with concurrent programming, such as testing a program with a few representative inputs.

Ben-David Kolikant (2004) describes learning concurrent programming in terms of entering a community of computer science practitioners. The focus of her analysis is on the utterances of high school students while they are solving a concurrent programming assignment and the different perspectives on programming they represent. Specifically, she finds that the students, who have no programming experience but do have experience in using computer software, initially approach the concurrent programming assignment from a user's perspective, in which only the program behaviour seen through the user interface is taken into account. While one of the two students on which the analysis focused was able to switch to a programmer's perspective, allowing her to reason about synchronisation goals and possible interleavings and to systematically form a correct solution, the other continued to maintain a user perspective.

Hughes et al. (2005) state that even though many articles have been published on the subject of teaching concurrent programming, their review of two important forums for computer science education research uncovered no articles whatsoever on the subject of evaluating students' learning of concurrent programming. They argue that there is a need for quantitative empirical evidence.

4 Phenomenography

Marton (1981) notes that the world can be studied in terms of two different perspectives that he terms *first-order* and *second-order*. The first-order perspective involves examining and making statements about selected aspects of the world (*phenomena*), while the second-order perspective involves examining and making statements about how people experience these phenomena.

Marton argues that although educational research often focuses on the first-order perspective, the second-order perspective can also be fruitful in educational research. He notes further that second-order knowledge cannot normally be derived from first-order knowledge; we have no effective way to deduce how different people think about the world from what we know about it.

Marton also points out that people perceive concepts in many different ways. Booth (1992) describes, for example, the different ways in which students perceive recursion. Eckerdal & Thuné (2005) provide a recent example from the computer science domain: novice Java programmers perceive objects and classes in various ways.

Marton (1981) continues by describing the second-order research approach, which he terms *phenomenography*, as "research which aims at description, analysis, and understanding of experiences" and states that its focus is on understanding the variation in these experiences. The outcome of a phenomenographic research project is thus a set of *categories of description*, where each category describes a qualitatively different way in which a phenomenon is understood in a cohort (Marton & Booth 1997).

Berglund (2006) describes the process of phenomenographic research in computer science education as consisting of a data collection phase and an analysis phase. In the data collection phase, the researcher interviews students about the phenomenon under investigation. The students are chosen with the intent of getting a diverse sample, in order to get a rich variation in their experiences of the phenomenon. Similarly, the interview must allow the student to express his understanding of the phenomenon of interest in many different ways. The interviews are then transcribed for analysis, during which the researcher looks for quotes that illuminate the students' various understandings and classifies quotes into categories of description. The analysis phase is typically iterative, with the tentative categories changing repeatedly as the researcher refines his analysis.

5 The study

In this section, we present the setting, how the data were collected and how the analysis was performed.

5.1 Setting

The students in this study participated in the Concurrent Programming course at Helsinki University of Technology during the autumn of 2006. All assignments were to be done in Java (version 1.4 or earlier). The assignments are described in more detail on the course home page[1]. Students could choose to do the assignments alone or in pairs; in either case, each group of one or two students submitted one solution that was graded the same way irrespective of the size of the group. As the students were required to submit their solutions through a WWW form that compiled their code, all submissions were valid Java programs.

The students were initially required to submit only their Java source code. In the event that their solution was rejected, they were required to submit corrected program code and reports explaining the reasoning behind the erroneous code and the steps they took to correct it.

In the first assignment, *Trains*, the students are given a simulated train track with two trains and two stations. Their task is to write code that drives these trains safely from one station to another using semaphores to co-ordinate the trains' movement.

The second assignment, *Reactor*, involves the Reactor pattern and its application to a simple multiplayer Hangman game. The students' task is to implement, using the synchronisation primitives built into Java, a dispatcher and demultiplexer for classic Java I/O, and to use this to implement a simple networked Hangman game that uses this Reactor pattern implementation.

The interviews focused on the third assignment, *Tuple space*, in which the students implement a tuple space using Java synchronisation primitives and use this to construct the message-passing section of a distributed chat server. The students' message-passing code communicates with the rest of the chat server system using method calls; a simple GUI front-end to the system is provided to the students for testing. The tuple space version in this assignment is a simplification of the original version: the `read()` operation has been removed, as it can be replaced

[1] `http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml`

by an `in()` followed by an `out()` with the recently removed tuple without compromising the correctness of the operation. For consistency with Java naming conventions and clarity, `out()` and `in()` have been renamed `put()` and `get()` respectively.

5.2 Interviews

The first author conducted interviews with eight selected students regarding the Tuple space assignment of the Concurrent Programming course of Autumn 2006. The interviews were conducted between the announcement of initial submission results and the resubmission of failed assignments. The focus of the interviews was on the development process, especially the students' reasoning behind their design.

Twelve groups of students (nine students who did the assignment alone and three pairs who collaborated on the assignment) were selected for interview based on the assessments of their initial submissions for the third assignment. In order to maximise the variation of experiences based on the information available to us about the students, we chose groups with different types of problems with their code, as determined by the teaching assistant who graded the assignments. Ten out of 31 groups that failed the (initial submission of the) assignment and two out of 24 that passed the assignment (on their first try) were chosen and invited to an interview. Out of these groups, seven of the failing groups (six single students and one pair) agreed to participate.

The interviewees were allowed to choose the language of the interview: Finnish, English or Swedish. Five groups were interviewed in Finnish. Two East European students were interviewed in English, their primary language of instruction at our university. The two participating students who worked together on the assignment were interviewed together.

The interviews were semi-structured, i.e. they were in the form of a conversation using a set of prepared questions as conversation starters, and lasted from 30 minutes up to almost an hour. The questions were about tuple spaces, the design decisions made by the students in solving the assignment, their approach in determining whether their solution was satisfactory, and problems found by the students or the teaching assistant.

All of the interviews were recorded using a single table-top microphone and transcribed by the interviewer. The interviewer also wrote down the main points of the interview directly after the interview. These recordings and notes, as well as the code and documents submitted by the students and the teaching assistant's assessments of the students' submissions, form the source material.

5.3 Analysis

The analysis was done by the first author in discussion with the second author. Specifically, the authors first discussed the contents of two interview transcripts and then the iterative phase of the analysis was performed. In each iteration, the first author read through the transcripts looking for relevant quotes and formed categories based on these, building on the results of the previous iteration. The categories were grouped into outcome spaces by the issue they describe. The second author then examined these categories and made suggestions on how to improve them. The resulting categories from the last iteration are presented in the following section.

In the first iterations, the analysis focused on finding as many quotes as possible that illustrated ways in which the interviewees understood concurrent programming and approached the assignment. Initially, quotes were grouped together if they essentially said the same thing (for example, tuples are, or are represented as, arrays). Then they were grouped together into tentative categories representing similar understandings of a phenomenon, which were grouped into tentative outcome spaces based on the phenomenon being discussed.

The categories changed in many ways during the analysis process. Starting from the third iteration, the emphasis of the analysis shifted to refining the preliminary categories. In some cases, only a few quotes regarding a phenomenon were found, in which case the data were deemed insufficient for further analysis. By the fourth iteration, only two outcome spaces remained in consideration as relevant for this paper; they are presented in the following section.

6 Results

In this section we present the outcome spaces of our phenomenographic analysis. In Subsection 6.1 we present the different *understandings of tuple spaces* that we found. In Subsection 6.2 we present the different *understandings of developing and debugging a concurrent program.*

Quotes are used to illustrate the categories. In these, the interviewer is denoted *Int* and the interviewees are assigned, to preserve their anonymity, the names *Evgeniy* and *Elena* (interviewed separately in English), *Filip*, *Fabian*, *Fritjof* and *Frans* (interviewed separately in Finnish) and *Freja* and *Fredrik* (interviewed together in Finnish). The quotes from the interviews in Finnish have been translated into English by the interviewer.

6.1 Tuple space

In this subsection, we present the different ways in which tuple spaces are described by the interviewees. This is summarised in Table 1.

6.1.1 Specification

In this category, a tuple space is understood as a *specification*, i.e. as *a set of operations and how their inputs and outputs relate.*

An example can be found in the following extract from the interview with Evgeniy:

> **Int:** Could you briefly explain what a tuple space does?
>
> **Evgeniy:** There are two operations, to put a tuple in and... to... say, get a tuple in with a pattern... Uh, to get a matching tuple.
>
> **Int:** If you have several matching tuples, which one do you get?
>
> **Evgeniy:** Whichever, practically.
>
> **Int:** And, if there's no match?
>
> **Evgeniy:** Then, the execution suspends until there is one.

He explains what a tuple space is by referring to its definition. Fabian gives a similar view:

> **Int:** So, what's the similarity [in the tuples] between different machines, given that you can't refer to the same variable?
>
> **Fabian:** So, like, there are similar parts. There are like, the same, for example, they've been marked... The ones that, like, fit the pattern... It should match. And there are certain parts that match it and certain parts can be anything. Yeah, that's how

	Label	What is the tuple space described as?	What is in focus?	Framework
1	Specification	Operations on tuples	The properties of the operations	-
2	Implementation	Data structures and code	How a tuple space implementation works or could work	Part of a program
3	Usage	A tool to achieve a specific subgoal in a program	What a tuple space can be used for in a program	A program
4	Evaluation	A better way of co-ordinating distributed systems	The advantages of using the tuple space	Other communication and distributed data storage mechanisms

Table 1: Categories of tuple spaces

you get. Then you can directly mark what belongs directly, put some sort of identifier at the start or... that way, what you want to get from there. The message has the same identifier, then.

This statement is still based on the definition, but Fabian concentrates his explanation on the concept of a pattern.

We have in this category encountered an understanding that resembles that of a textbook definition, here exemplified by the following quote from Andrews (2000):

> A process extracts a data tuple from TS by executing `IN("tag", field`$_1$`, ..., field`$_n$`);` Each `field`$_i$ is either an expression or a formal parameter of the form `?var` where `var` is a variable in the executing process. The arguments to `IN` are called a template. The process executing `IN` delays until TS contains at least one tuple that matches the template, then removes one from TS. (ibid, p. 335)

Here, both Andrews and the students explain tuple space operations in terms of the operations on the tuple space and the inputs, outputs and delays of these operations.

Fritjof prefers to explain tuples in terms of programming language constructs rather than as abstract groups of values:

> **Int:** Could you describe this tuple space? Like, what you put in it, what it does, what you get from it, sort of on that level?
>
> **Fritjof:** Yeah. Uh... I don't have a fancy understanding of it, or one that is... necessarily entirely correct, but I'd say it's just like a set or a space containing unordered items. So, I don't know about these tuples, but I'd imagine, or I like to think of them as sort of arrays of some sort of elements, so, for example, a tuple is some n-element table in there.

Here, Fritjof uses programming language constructs such as arrays, but is still describing the tuple space in terms of its interface (in this case, the data format used to communicate with it).

Fredrik explains pattern matching in the tuple space in similar terms:

> **Int:** Yeah, how do you choose what to get, for example?
>
> **Fredrik:** It's quite...
>
> **Freja:** Isn't it kind of like getting with parameters that are 'identifiers' for these tuples? So they sort of have an identifier.
>
> **Fredrik:** Right, right, right, and, the amount of fields or attributes and... also null values.

The statements of Fritjof and Fredrik are inspired by the requirements of the assignment, which provided a very specific definition of tuple space operations and Java-specific definitions of how tuples and patterns are represented:
`public interface TupleSpace`

- `public String[] get(String[] pattern)`

  Remove and return a tuple (an array of entries) matching `pattern` (which may not be `null`) from tuple space. Block until one is available. A tuple matches a pattern if both have the same amount of entries and every entry matches. A `null` entry in the pattern matches any object in that entry in the tuple. Any other object `p` in the pattern matches any object `t` in the corresponding entry in the tuple for which `p.equals(t)` (i.e. contains the same character string). If several matching tuples are found in the tuple space, any one of them may be returned.

  The returned tuple must have exactly the same textual contents as the tuple that was put (contain the same amount of `String` objects as the original and each `String` equals the `String` in the same position in the original), but may be a different array object and may contain different `String` objects).

- `public void put(String[] tuple)`

  Insert `tuple` in tuple space. `tuple` is an array of any length greater than zero and is not `null`. `tuple` may not contain `null` values. Tuples stored in the tuple space must remain unchanged as long as they are in the space.

Thus, the two interviewees discuss what a tuple or a pattern is, based on the definition given in the assignment[2].

In this category, the tuples per se are in focus. Different aspects of them can be highlighted, such as operations on them (Evgeniy) or the structure of the tuple (Fritjof) or both (Fabian). The specification might have its roots in the theoretical specification (Fabian) or from the assignment the students were to solve (Fritjof and Fredrik).

According to phenomenographic theory, when someone experiences something, some aspects of the experienced phenomenon stand out in the fore, while other aspects of it or other contextually related phenomena reside in the background (Marton & Booth 1997). However, this category shows a slightly different structure: as the tuples are seen in isolation,

[2]http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml

in a decontextualized manner, they are experienced not against any background, but as atomic objects in their own right.

6.1.2 Implementation

In this category, a tuple space is understood *as it is implemented*.

Let us listen to Fritjof, for example:

> **Int:** . . . and the get operation does what?
> **Fritjof:** So, if you call `get()` with a certain pattern, something, you want from there a specific tuple; it looks through the tuple space for such a set or item. If it finds it, it returns it directly, immediately. If it doesn't find, then it actually waits for someone to put an item there with the `put()` operation.

This way of understanding tuple spaces also allows the large-scale structure of the implementation to be described, as in Frans's statement:

> **Int:** OK, right, and how does this transfer affect the tuples, then?
> **Frans:** Those tuples are somewhere in a central place, so that means that if you get something from there then it isn't there anymore. It should, uh, take into account that it. . . If there's some tuple there and somebody gets it from there, then nobody else can get it from there before that somebody has returned it there.

His statement illustrates a problem with this way of experiencing tuple spaces: the implementation need not be the same everywhere.

Fredrik starts his description of a tuple space in the following way:

> **Int:** So how does it [the tuple space] work, then? How is it used?
> **Fredrik:** It's a data container in which it has been ensured that you don't read and write to it at the same time through synchronisation.

Fredrik's statements, that seemingly express a high-level understanding, actually reflect his own implementation of tuples in the assignment. There, he relies on a single lock (implemented using the Java `synchronized` keyword) that ensures that only one operation at a time is performed on his tuple space. This, in turn, ensures that the tuple space operations behave atomically, as specified.

This category extends the previous, as the implementation is written to match a specification, or at least is written to achieve a goal. In any case, the externally visible behaviour (which the specification describes) can be deduced from the implementation.

6.1.3 Usage

This category describes a tuple space as *a data structure or a module that can be used as a part of a program in order to achieve a specific subgoal in a program.*

This is illustrated by Frans's answer, when asked to explain how using a tuple space affects the program:

> **Int:** OK, so they're [the tuple spaces] intended for distributed systems that may have a common space for many machines or processes, despite not having variables in common. How does this affect them?
> **Frans:** Um, what are you getting at? Apparently, with the help of the tuple space you can implement some sort of monitor or something, with which you can. . .

He states that synchronisation is the purpose (or at least one purpose) of using tuple spaces. Clearly, this statement refers to usage of the concept within a program.

Fritjof explains how his chat system uses a tuple space to get unique message numbers over several distributed processes:

> **Int:** So, how does it [your implementation] work when it's on several machines?
> [. . .]
> **Fritjof:** I put these specific tuples there, that always stay the same, so, like, even though the message counter, that is. . . that is, that way you keep track of those messages that are put in the tuple space, so we have this single message counter tuple there, and with the aid of that, uh, the remote machines sort of synchronise their functioning, so the counter is fetched from there. One machine gets the counter, then another machine, even though it's trying to get the message counter tuple at the same time, it doesn't find it there. And. . . And when the first machine has processed the tuple, got the value from there, it puts the tuple back in there and the other machine can then get it from there.

In this case, he is describing the *implementation* of a chat system that *uses* a tuple space for communication. The characteristic of this category is how a tuple space is used, or its purpose in a program. This implies a broader perspective than that of the previous category, since the tuple space must be seen in the context of a program for its usage within a program to be seen.

6.1.4 Evaluation

Here, the tuple space is seen in terms of the *advantages of using it in contrast to other data structures or message passing mechanisms.*

When asked how using a tuple space affects the programs using it, Elena answered:

> **Int:** OK, so if you have a distributed environment here, where you have one pro. . . two different programs that might be in completely different machines with a tuple space, and, uh. . . ?
> **Elena:** It makes, uh, the communication between them; it makes it very much easier, so, um, between different. . . ah, implementations, they can communicate with each other by way of this pool.

Filip compares the tuple space with a semaphore:

> **Int:** Could you explain what a tuple space is; how it behaves in general. . . ?
> [. . .]
> **Filip:** It's kind of like an improved version of a semaphore, so, in a semaphore, like, you've got to know in advance what the semaphore is connected to, but the tuple, you can attach information to that. But it. . . It's like sort of. . . A semaphore is a special case, you can also use a tuple in such a way that it either has a flag set or not.

Here, he is contrasting the tuple space with a semaphore, and noting that a semaphore can only indicate that something has happened (typically, that a resource is available), while a tuple can contain additional information (such as the contents of a message).

The purpose of using tuples as building blocks of a program, which was in the fore of the previous category, is taken for granted here. Instead, the benefits of using tuple spaces are seen in relation to other ways of communicating.

6.2 What does it mean to write and debug a concurrent program?

In this subsection, we present the different ways in which the interviewees understand the process of developing and debugging their program. This is summarised in Table 2.

6.2.1 Implementation

In the first category, writing and debugging a concurrent program means *making it run*; the *coding* itself is the focus.

The programming of a complex sequence of events can be experienced in this way. Fritjof explains his message-writing implementation:

> **Int:** So, how did you fix it [a race condition in using the message counter]?
> [. . .]
> **Fritjof:** Anyway, the idea is that, like, in that `writeMessage()`, uh, I right at the start call, uh, that it fetches the message counter tuple. . . and its. . . method that it's fetched with, it only removes the tuple from there. . . the counter tuple, so, it, like, fetches it for itself. . . So, this time, it doesn't, like, do those `put()`/`get()` operations in the same method, so it just takes them from there, and then after that. . . After that it, uh, writes the, uh, client's message, or makes a tuple that is the client's message, and uses the counter that it fetched. And, uh. . . The message is always put in the tuple space. After that, uh, the message ID is incremented, which, of course, is for every, uh. . . chat channel separate and not until that is done, finally, the tuple is put back in the tuple space.

Java constructs (e.g. methods, calls, incrementation) and the roles in which they are used (e.g. tuples, counters) are taken for granted and constitute the building blocks from which his explanations are built.

While Fritjof discusses programming, we can hear a similar discussion from Evgeniy when he discusses debugging:

> **Int:** In what did the first attempt. . . ?
> **Evgeniy:** The first attempt uh. . . had just the backlog of messages in the tuple space. . . Uh, where, with a, I th. . . I don't remember was it counters or something where channel listeners would pick a tuple from the space and return it until the counter is zero and then discard it instead, but, uh, for some reason; for some obscure reason I don't understand, still, it didn't work, uh, when there were more than one channels, uh, more than one listeners or with, with load it started to go all bad and the. . . tuple started to disappear.

Evgeniy explains how an early version of his program misbehaved in low-level terms. Although he does not use language containing Java constructs, his description is worded, as Fritjof's above was in part, in terms that correspond to specific Java constructs. For example, Evgeniy's counters are integers stored as fields in tuples of a specific form.

Elena also takes her point of departure in the program itself, when asked to explain her chat system design:

> **Int:** Was this design the first one that occurred to you or did you consider some other way of doing it?
> [. . .]
> **Elena:** The basic idea, however, I can, so when, ah, when, uh, when, uh, the method `writeMessage()` of the server is called, what it does is that it gets the. . . the number, because there's a tuple which keeps the maximum number of servers.
> [. . .]
> **Elena:** New, newcoming servers see the. . . uh, get the tuple and update it with an increased number. . .

In this quote, Elena specifically takes her point of departure in the program execution at the level of the programming language when she describes how data must flow between different servers.

The three students whose quotes have served to illustrate this category have all explained writing and debugging of programs in terms of the implementation of the programs. The core of this category is the execution of the program, and descriptions at the level of Java code, regardless of whether the students worded their explanations in Java terminology. This is seen against a background delimited by the language features that can be used in a certain situation. In other words, the reasoning does not extend the program and its execution in any way, other than assigning roles and purposes to the Java language constructs.

6.2.2 Solving technical problems

In this category, writing a concurrent program is experienced as *solving technical problems*.

Let us listen to Filip, for example:

> **Int:** So, how is this information transferred [between machines], then, roughly?
> **Filip:** Well, it remained kind of unclear to me how it would be done in a practical application, because, because, uh, one should, like, be careful that the tuples are always the same on all machines and then, if somewhere an acquire is done, then the information is transferred to them all before they can do anything at all [. . .] to the corresponding tuple.
> [. . .]
> **Filip:** Like, that all the, sort of, servers have to know the same, like, tuple space. They've got to have all tuples known to everyone.

He raises the issue that the tuple space is a data structure intended for distributed computing and that the different computers involved must co-operate to ensure that operations performed on the space on one computer are visible on all other computers. He discusses this in terms of a technical system, that is, he describes 'what happens' and 'how it functions' in a

	Label	What is developing and debugging described as?	What is in focus?	Framework
1	Implementation	Writing and debugging code	The code and its execution	Relevant programming language constructs
2	Solving technical problems	Finding solutions to a series of technical problems	Central ideas of concurrent programming	The program, seen as a technical entity
3	Producing an application	Finding solutions to real-life problems	What users need from the program	Context in which program is used

Table 2: Categories of developing and debugging

different way from that expressed in the previous category, where the focus is on language constructs as primitives.

Similarly, when asked to elaborate on his reasons for choosing a particular chat system design, Fritjof notes:

> **Int:** Why'd you choose this particular solution?
> **Fritjof:** ... a 'send copies to everyone'-style solution first came to mind, but, uh, then it... Then it came to me that it isn't really clever to do it this way, so... So, one, only one copy should be stored at a time.

Frans takes his point of departure in similar concepts:

> **Int:** Do you have any idea what messages it can trans... leave undeleted?
> **Frans:** When each active listener is sent the messages, could it be that someone... some listener, some active listener leaves before it's read all the messages sent to it?

When asked why his system may incorrectly leave some messages indefinitely in the tuple space (a form of memory leak), Frans answers in terms of messages being sent to listeners, the large-scale behaviour of the system, rather than the underlying tuple space operations.

Here, we have seen a way of experiencing writing and debugging programs in which the task is seen as solving a series of technical problems. This is viewed and expressed in terms of general and central ideas of concurrent programming and the system that is constructed. This can be contrasted with the previous category, in which the basic entities are the programming language and the constructs expressed in it. Thus the current category has a broader perspective.

6.2.3 Producing an application

In this final category, the programming task is understood as *solving problems relevant for a usage context.*

Filip demonstrates this viewpoint when responding to the teaching assistant's complaint that his solution does not allow messages to be repeated or be empty:

> **Int:** So you mean you've planned your solution to sometimes duplicate messages and then compensated by deleting them later?
> **Filip:** Yeah, I figured the duplicate removal didn't hurt, especially since it doesn't happen randomly, but when a new user joins the channel. Then, uh, the empty messages, that's just that I've assumed that you don't want to put empty messages. It's, both in the transmission and reception, been tested whether an empty message has come, but this was also an error that led to failing the assignment.

As opposed to the previous categories, the technical concepts here become tools for solving a real-life problem. The student is no longer working to implement a specification; he is trying to meet the needs of the users. The background is therefore no longer a purely technical context (the program); it is the real-world situation for which the program is intended. In addition to the system-level technical concepts of the previous category, the basic entities now include user requirements and desires.

7 Discussion

In this section we will examine the categories presented in the previous section from different perspectives. First we will discuss the meanings of the different categories from an educational point of view. We will then discuss what the categories mean for our long-term research.

7.1 Tuple spaces

Bloom's taxonomy of educational objectives (Bloom 1956) is widely used in higher-education course design to ensure a proper balance between rote learning and high-level skills such as synthesis and evaluation. However, its applicability to computer science teaching is debatable, as the goal of computer science is often perceived by teachers to be application. (Johnson & Fuller 2006)

Johnson & Fuller (2006) therefore propose a revised Bloom taxonomy which retains *knowledge*, *comprehension* and *application* in their original order, places *analysis*, *synthesis* and *evaluation* as equals above the lower three and adds *higher application*, application informed by analysis, synthesis and evaluation, at the top. The revised taxonomy is particularly relevant when computer science is approached from an engineering perspective where application is clearly the goal. This motivates a comparison with our tuple space categories, in which students understand a concurrent programming concept in terms related to different skills and tasks.

The specification category is essentially knowledge; the students are explaining the tuple space description given to them in the textbook, lectures or assignment specification.

In the implementation category, the students have constructed their own implementation, which at least requires application of the specification and concurrent programming in Java.

The usage category is another form of application, except here the tuple space knowledge is being applied to achieving a program's goals using a tuple space. Finally, the evaluation category clearly corresponds to evaluation.

The tuple space categories therefore span a large portion of the revised Bloom taxonomy, although analysis and synthesis do not appear. Although lack of evidence is not evidence of lack (especially since only students who failed the assignment were interviewed), this does raise the question of whether the teaching of concurrent programming can be improved by encouraging students to analyse and synthesise.

7.2 Development

Software engineering emphasises ways of managing complexity and quality that rely on different perspectives on the software that is being developed. The categories of developing that we found are similar to several of the different views needed in many common software development processes.

The implementation categories of both developing and tuple spaces are obviously necessary for practical software development, in which formulating an implementation in a programming language is essential. The solving technical problems category corresponds to design. In this assignment, what the students are performing is essentially module design, as the specification the students are provided with is more or less a finished architecture design. This specification is also seen in the specification category of tuple spaces. The design of the chat application also involves usage of tuple spaces; evaluation is not necessary in the assignment, as the use of tuple spaces was mandated.

These first two categories of developing can also be considered facets of what Ben-David Kolikant (2004) calls the programmer's perspective, which includes reasoning in terms of both the concurrency model and the implementation.

As the assignment is to perform the low-level design and implementation of parts of a system with clearly specified requirements, we did not expect any consideration of the user of the system. The application production category, which is suited for requirements analysis, is therefore unexpected in this study. As seen in Subsubsection 6.2.3, it can become a hindrance in programming assignments such as the one in this study. Adherence to the specification is considered by the teaching staff to be the goal of the assignment. This leads to problems when the student moves beyond the specification to address perceived user requirements instead. This also demonstrates that having a skill or understanding is not enough; the student must also learn to use it in the appropriate context.

Each one of these categories is suitable for some task in software engineering. Taken as a whole, these three categories more or less span the perspectives necessary for a full software development process. This suggests the idea of teaching students requirements analysis for concurrent programs instead of keeping the teaching and assignments on the design and implementation levels. This could also help students understand the appropriate situations in which to use each skill.

8 Conclusions

In this paper we present different categories of understanding tuple spaces and developing a concurrent program. We describe the relationship between these categories and the skills we want students to develop. We then discuss how this information can be used in developing concurrent programming education, by exposing students to different views of the same phenomena. We also discuss the use of this study in our long-range research in supporting concurrent programming, first by helping develop a model of how bugs are introduced in concurrent programs, and second by developing software to display information about these programs in a fashion consistent with the programmer's understanding.

8.1 Long-term research impact

In the previous section we concentrated on an educational perspective on the results of this study. We will now consider our results in terms of our long-range research questions, as described in Section 2.

Different errors can be the result of completely different ways of thinking. In some cases, such as the application production category described in the previous section, approaching a problem from the wrong perspective may lead to erroneous conclusions. In other cases, the nature of the errors depends on the perspective or task at hand. Thus understanding how the programmer is thinking is important in finding ways to prevent errors from being made as well as determining the errors to look for in verification. For example, if a programmer misunderstands the requirements or specification of a system or module, he will be also be testing according to his erroneous understanding of the requirements. One way to address this is to include test cases in the specification, providing both clarifications to the programmer and test cases that are not dependent on his understanding.

This is one way in which this study supports our long-range goals: as software defects are the results of programmer error, i.e. incorrect thinking, it is necessary to understand how programmers think in order to construct a model of errors. For example, goal-plan analysis (Spohrer et al. 1985) provides a way to identify and categorise bugs based on a model of the problem-solving process. For each goal, there are one or more plans for achieving it, each with several subgoals; this is expressed as a goal and plan (GAP) tree. Spohrer et al. (1985) infer the GAP tree from the programs. Knowledge of how programmers understand their development process is both a source of possible plans and empirical support for the plans identified from the programs.

Another way in which the results of this study can be used in our research is in designing useful visualisations of programs and their execution. Debuggers traditionally focus on code, as in the implementation category. However, the solving technical problems category suggests an alternative perspective on debugging: that it would be useful to provide supporting tools, such as execution visualisations that show program behaviour in ways that support the user's understanding. This could be done by allowing the user to group together parts of the code or execution to correspond to his understanding, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. (1993). The tool would then visualise the behaviour of the program in a fashion closer to the programmer's view. For example, if the programmer sees his program as a set of communicating entities, the tool should be able to show him the communication between these entities and the relevant aspects of their state even though this state may be spread out over several objects, and part of the communication is implicit in locking mechanisms.

8.2 Future analysis of the data

The semi-structured format of the interviews allowed the interviewees to express themselves on a wide range of subjects related to concurrent programming and the associated teaching and assignments. One possible topic for future analysis of these interviews is the types of errors students make and the factors that contribute to them. Another is how students approach and understand the learning of concurrent programming and the testing and debugging of concurrent programs. It is also possible to examine the categories presented in this article in greater detail, examining different conceptions and misconceptions within each category.

References

Andrews, G. R. (2000), *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley.

Ben-Ari, M. (2006), *Principles of Concurrent and Distributed Programming*, second edn, Pearson Education.

Ben-Ari, M. & Ben-David Kolikant, Y. (1999), Thinking parallel: The process of learning concurrency, *in* 'Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education', Cracow, Poland, pp. 13–16.

Ben-David Kolikant, Y. (2004), 'Learning concurrency as an entry point to the community of computer science practitioners', *Journal of Computers in Mathematics and Science Teaching* **23**(1), 21–46.

Berglund, A. (2006), 'Phenomenography as a way to research learning in computing', *Bulletin of Applied Computing and Information Technology* **4**(1).

Bloom, B. S. (1956), *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*, Addison Wesley.

Booth, S. (1992), Learning to program: A phenomenographic perspective, Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden.

Eckerdal, A. & Thuné, M. (2005), 'Novice Java programmers' conceptions of "object" and "class", and variation theory', *SIGCSE Bulletin* **37**(3), 89–93.

Gelernter, D. (1985), 'Generative communication in Linda', *ACM Transactions on Programming Languages and Systems* **7**(1), 80–112.

Holzmann, G. (1997), 'The model checker Spin', *IEEE Trans. on Software Engineering* **23**(5), 279–295.

Hughes, C., Buckley, J., Exton, C. & O'Carroll, D. (2005), 'Towards a framework for characterising concurrent comprehension', *Computer Science Education* **15**(1), 7–24.

Johnson, C. G. & Fuller, U. (2006), Is Bloom's taxonomy appropriate for computer science?, *in* A. Berglund & M. Wiggberg, eds, 'Proceedings of 6th Baltic Sea Conference on Computing Education Research, Koli Calling', Uppsala University, pp. 120–123.

Marton, F. (1981), 'Phenomenography — describing conceptions of the world around us', *Instructional science* **10**, 177–200.

Marton, F. & Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associates.

Pears, A. N. (1995), Using the DiST simulator to teach parallel computing concepts, *in* 'International Forum on Parallel Computing Curricula', Wellesley, Massachusetts.

Price, B. A., Baecker, R. M. & Small, I. S. (1993), 'A principled taxonomy of software visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.

Spohrer, J. C., Soloway, E. & Pope, E. (1985), 'A goal/plan analysis of buggy Pascal programs', *Human-Computer Interaction* **1**, 163–207.

Visser, W., Havelund, K., Brat, G., Park, S. & Lerda, F. (2003), 'Model checking programs', *Automated Software Engineering Journal* **10**(2), 203–232.