Description and Comparison of the Concurrency Concepts of Ada, Java, and CHILL

Project Alfa Core

(Studienarbeit)

Friedrich-Schiller University Jena
Faculty of Mathematics and Computer Science
Institute of Computer Science
D-07740 Jena, Germany

submitted by: *Peter Brömel*, born July 25, 1973 in Eisenach
Frank Ecke, born October 22, 1974, in Sondershausen

Supervisor: Prof. Jürgen Winkler (FSU Jena)

Jena, January 13, 1999

Abstract

In this paper, we will describe and compare the concurrency concepts of the languages Ada, CHILL, and Java. We shall cover basic topics such as definition, creation, and termination of actors, and we will then procede to examine the executional part of an actor. Following that is a treatise on direct and indirect actor communication—the latter leading us to the discussion of objects with coordinated access. In the final chapter, we try to give an overview of realtime programming and the special requirements for realtime systems.

The work of writing this paper was distributed between the two authors as follows:

- Chapter 0: Frank Ecke
- Chapter 1: Frank Ecke
- Chapter 2: Peter Brömel
- Chapter 3: Frank Ecke
- Chapter 4: Frank Ecke
- Chapter 5: Peter Brömel
- Chapter 6: Frank Ecke, except for the PEARL section (which was written by Peter Brömel)
- The Glossary: Peter Brömel

Table of Contents

0 Preliminaries

0.1 General Introduction to Concurrency

Programs written in traditional languages such as Pascal, Algol, C, Fortran, Cobol, or Modula-2 share a common property—sequentiality. The programs start executing in some initial state and enter, by obeying one statement at a time, subsequent states, until the program terminates. The path through the program may differ due to variations in the input but for any particular execution of the program, there is only one path. For the remainder of this paper, this path shall be known as *thread of control.* From what we have mentioned so far, we can conclude that sequential programs possess a single thread of control.

A concurrent program, on the other hand, may contain multiple threads of control, each of which may be independent and executed separately. Roughly speaking, such programs have the potential to do "several things at once". For the sake of illustration, we can imagine a concurrent program to consist of several parts that can be executed in parallel.

It is, by now, well understood that concurrency is of significant importance. The most apparent reason is mankind's desire to model the real world, which is the largest concurrent system we know. Embedded systems, air traffic control, avionics systems, industrial robots, and engine controllers—to name a few examples—are inherently parallel. To use the power of computers in these areas, the software must control the concurrent activities and allocate hardware resources appropriately.

In addition, recent progress in multi-processor hardware design calls for efficient exploitation of these new possibilities. The ability to map a concurrent algorithm directly—by means of a concurrent programming language—onto the hardware is clearly desirable. Increasing reliability and fault tolerance are further motivations for concurrency.

"Some might argue that such matters are the concern of the operating system and are better done by calls from an otherwise sequential program. However, built-in constructions provide greater reliability, general operating systems do not provide the control and timing needed by many applications, and every operating system is different." (Barnes, 1996, [p. 393])

(Burns, 1998, [p. 22]) cite Ben-Ari for the clarification of our subject: "Concurrent programming is the name given to programming notations and techniques for expressing potential parallelism and for solving the resulting synchronisation and communication problems. Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming. Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details."

We do not, in this paper, distinguish between true parallelism and pseudo-parallelism. We define two actions to be *concurrent*, if they are overlapping in time. That is, given two actions, A and B, there is at least one point in time at which both A and B are executed. Two programs are *concurrent*, if they contain concurrent actions.

Note that it is not necessary to consider an "arbitrary" number of concurrent actors. Although in reality there will almost ever be more, it suffices to study cases involving two concurrent actors only. All problems that arise in such cases can be easily generalized to any number of actors. In fact, a larger number does not give rise to new aspects.

Figure 0.1 shall serve as an illustration for actions that are concurrent (see below for how to read figures throughout this paper).

To avoid interference with the terms used in the three languages, we will, on the abstract level, use the term *actor* for entities that can be executed concurrently. We thus realize that an actor is an entity possessing its own thread of control.

In general, there are more actors than physical processors—we will stick to that.

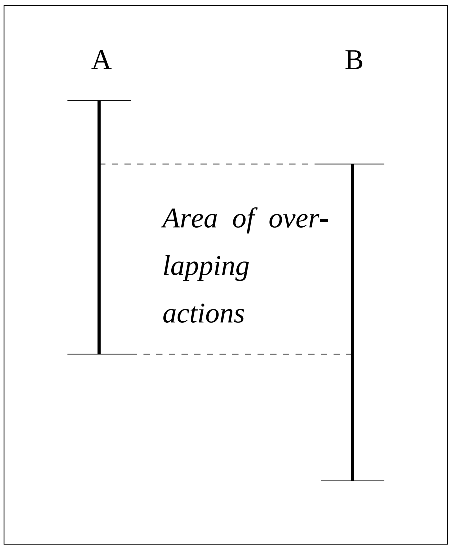

Figure 0.1: Concurrent actions

0.2 Problems and Pitfalls

Concurrent programming offers many advantages but, unfortunately, has pitfalls, too. Due to a possibly large number of concurrent activities, such programs tend to be much harder to design, understand, and prove correct. Psychology reveals that, albeit the human brain is capable of employing true parallelism otherwise, the very act of contemplating in the mind is singled threaded.

The use of concurrent programming techniques introduces problems that are completely unknown in sequential programming—multiple update, deadlock, livelock, race conditions, and starvation being examples of these. For the update problem, consider the following scenario: Assume that a globally accessible variable is to be updated. A set of otherwise unrelated actors is permitted to add to or subtract from this variable. The update might read as follows:

```
Read R1, Variable
Add Delta1, R1
Store R1, Variable
```

If there is only one actor doing the update, then nothing can go wrong, since at most one actor has access at any time. Several actors, however, might interfere. Suppose that `P1` starts updating by reading `Variable` into register `R1` and then adding `Delta1`, say 10, to register `R1`. But before `P1` can finish its update, it is preempted by `P2` which now reads the value into `R2`, adds `Delta2` (say 100), and is preempted by `P1` which now finalizes its update by writing `R1`'s contents back to `Variable`. As the last step, `P2` writes `R2` into `Variable`. Example 0.1 illustrates this.

Having successfully finished its update, `P1` now assumes that `Variable` has been increased by 10 whereas `P2` relies on an increase by 100. Clearly, `P2` is right, but how can `P1` know that its assumption is faulty? The correctness of the transaction relies on timing. This is *not* acceptable at all! A sufficient condition for the transaction to be correct is the demand that the two updates be non-overlapping in time, i.e., `P1` shall be capable to finish its transaction before `P2` is allowed to begin (or vice versa, with the roles of `P1` and `P2` interchanged).

Generally, n actions ($n \geq 2$) are said to be *atomic*, if they are non-overlapping in time. More formally, let A_1 and A_2 be two actions. Furthermore, let $Start(A)$ and $End(A)$ denote,

P1	P2	Variable
`Read R1, Variable`		100
`Add R1, 10`		
	`Read R2, Variable`	
	`Add R2, 100`	
`Store R1, Variable`		110
	`Store R2, Variable`	200

Example 0.1: Faulty update

respectively, the start and the end (in time) of an action A. Then

$$A_1 \text{ and } A_2 \text{ are atomic with respect to each other} \equiv A_1 \text{ precedes } A_2 \text{ or } A_2 \text{ precedes } A_1$$

A precedes $B \equiv \mathit{End}(A) \leq \mathit{Start}(B)$. Returning to our example from above, an atomic update is given in Example 0.2.

P1	P2	Variable
`Read R1, Variable`		100
`Add R1, 10`		
`Store R1, Variable`		110
	`Read R2, Variable`	
	`Add R2, 100`	
	`Store R2, Variable`	210

Example 0.2: Correct update

In order to prevent such and related problems from causing serious damage (data inconsistencies), a programmer must be aware of them and has to acquire the skills needed to tackle them appropriately. The intention of this paper is, therefore, to provide a detailed introduction to concurrency aspects of Ada, CHILL, and Java. Furthermore, we will compare the three languages with respect to concurrency.

There is a number of concurrent programming languages—our three are by no means the first (or last!)—, but we felt that Ada, CHILL, and Java are especially suited for approaches to concurrency. They all are high level languages and maintain the high level even in the field of concurrent programming. Low level concurrent programming languages are those that are used in *massively parallel computing*. Here, a programmer has—in addition to specifying the problem—to distribute the workload explicitly among the available processors. Those languages provide direct interfaces to the underlying hardware. Communication has to be explicitly programmed and relies heavily on the architecture. The result is (often) a non-portable program in which much of the effort is devoted to scheduling, dispatching, and communicating. Our languages encourage software engineering principles that have been found valuable and fostering the development process. And, last but not least, they are of practical importance[1].

0.3 How to read this Paper

This paper assumes familiarity with a high level imperative programming language and basic understanding of the problems in concurrency (such as multiple update).

We actually distinguish between the definition of actor types and the definition of actors based on actor types. That is, we consider type definition and variable/object definition to be different. Several actor objects may correspond to one actor type definition. A type is a template whereas a variable is an incarnation of a type.

Additionally, we would like to consider three distinct levels of "actor-related" occurrences: the static program structure, i.e., the program text, the dynamic program structure, i.e., actor objects at runtime, and the execution, i.e., a particular execution of an actor's sequence of statements.

Actor types only exist in the program text. Simply think of this existence as the lines of code that literally describe the type definition. Although there are languages (CHILL and Java being examples) that allow a type to be equipped with type specific components (often called *static components*) so that the type (descriptor) can be used to access these components, we should not confuse type and object. An actor type is considered to be a template from which objects/instances can be created. And, lastly, an actor type cannot be active; hence, there cannot be a particular execution of an actor type.

If we, then, use actor types to create instances of these types, we note that an actor definition in the program text is just that: the definition of an object of an actor type. The result of such a definition is, at runtime, an actor object; hence, we have reached the stage of the dynamic program structure. Finally, we can look at an actor under execution, i.e., we examine an actor's sequence of statements.

The transition from the static to the dynamic program structure is achieved by the creation of an instance of the defined actor. Typically, an object is created whilst the corresponding definition is elaborated or by means of dynamic object creation, using reference types (see Chapter 1 for how this is done in our three languages). To reach the execution stage requires an actor to become activated.

Should we ever mention an actor without specifying its context, we mean an actor object in the dynamic program structure. Note that this somewhat theoretical framework is filled step by step as we forge ahead. Table 1.1 at the end of Chapter 1 will then summarize what we have explained.

In the diagrams that are used for illustrating time-related behaviour, time increases from top to bottom and threads of control are shown as vertical lines. Solid lines are used to represent activity, whereas a dashed line indicates that an actor is dormant. A sample figure, Figure 0.2, is shown below; it illustrates the time-related behaviour of the faulty update scenario presented earlier in this chapter.

References to the literature are given in the form (Author, Year, [Location]), where Author is the primary author, Year the year of the current edition at the time of writing this paper, and Location can be a page number (or a range of page numbers), section number, etc. If not appropriate, Location is omitted.

Acknowledgements

We are deeply indebted to our supervisor, Prof. Dr. Jürgen Winkler, for his invaluable hints

[1] Modula-3, for example, guarantees the first two issues as well, but—as far as we know—nothing of significant importance has been programmed in this language

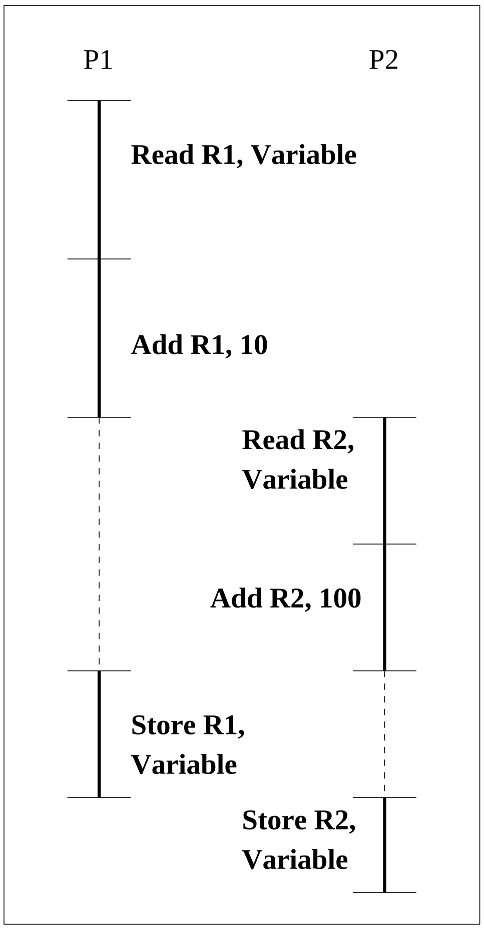

Figure 0.2: Faulty update

and for proofreading drafts of this paper. He has never got tired of harping on the subtle points. We would also like to thank S. Tucker Taft and Robert A. Duff for explaining the fine details of the Ada Standard. Furthermore, the folks on `comp.lang.ada` should not be left out for they have contributed significantly to the discussions about Ada-related topics. Last but by no means least, we express our gratitude to all the people whose books we have used as input to this paper. A corresponding list can be found in the bibliography section at the end of this paper.

The authors can be contacted for comments, additions, corrections, gripes, kudos, verbatim copies of this paper (Plain-TEX, LATEX, DVI, POSTSCRIPT), etc. by e-mail. The addresses are:

{`broemel, franke`}`@informatik.uni-jena.de`

Peter Brömel and Frank Ecke
Jena, 1998