Licentiate Thesis Proposal

An Operational Semantics for Parallel Execution of Re-entrant PLEX

Johan Erikson
Department of Computer Science and Electronics
Mälardalen University,Västerås, SWEDEN
johan.erikson@mdh.se

June 20, 2005

Abstract

A large class of legacy software systems, developed and maintained over many years, can also be termed sequential software systems in that independent parts of the system requires exclusive access to shared data during its entire execution. This requirement originates from design decisions on non-preemptive execution, and when the underlying architecture is a single-processor one, this is sufficient to protect the shared data. The problem arises when this architecture is to be replaced by a multi-processor ditto; since different tasks (still executed in a non-preemptive fashion, but on different processors) now may access, and update, the same data concurrently, non-preemptive execution does not protect the shared data any longer.

To the above problem, we propose a solution based on a program analysis that can decide when parallel execution of the current software is safe in the sense that the parallel execution does not result in data interference. As a formal basis for such an analysis, the formal semantics of the language in question has to be considered.

This thesis presents an operational semantics for the language PLEX, used to program the AXE telephone exchange system, in which the above mentioned properties are found: independent pieces of software, executed in a non-preemptive fashion, together with unprotected, shared data.

1 Background and Motivation

Sequential software consists of different, and independent, parts that requires exclusive access to common data during its entire execution. The exclusive access is due to early design decisions on that the independent parts of the system should be executed in a non-preemptive fashion, which means that there is no need to protect the shared data, as long as a single-processor architecture, in which the independent parts of the software are sequentially executed, is used. We can view a real-time system in which tasks/processes, on the same priority level, executes in a non-preemptive fashion, as an instance of a system that contains sequential software (and according to this relation, we will use the term task for the independent pieces of the system).

While complex legacy software systems, developed, and maintained, over many years, (such as telecommunication systems) may consists of large amounts of non-preemptive, sequential software, there is a development towards different forms of parallel hardware. Examples of such architectures are Symmetric Multiprocessors (SMP), Chip-Multiprocessors (CMP), and Simultaneous Multi-Threading processors (SMT), i.e., CPU's containing several processors (CMP), or processors that can execute several threads in parallel (SMT).

Sequential software could be executed on serial, as well as on parallel hardware, as long as the execution (on the parallel hardware) is restricted to sequential execution, see Fig. 1 (b), but problem arises when one wants to release the restrictions and execute several tasks with the same priority in parallel, thereby taking full advantage of the parallel hardware. At this point, the non-preemtive execution does not protect the shared data any more, since tasks that are executed on different processors may access and update the same data concurrently. The question is now: *How is the system to be parallelized?* By simply moving the software from the old architecture to the new, it is most likely that programs will break due to concurrent access of shared data, Fig. 1 (c).

A naive solution would be to re-design and re-implement the system, but since a legacy software system usually contain several million lines of code, this solution is infeasible due to the size of the system. However, since data is shared, access to it must be synchronized to avoid an

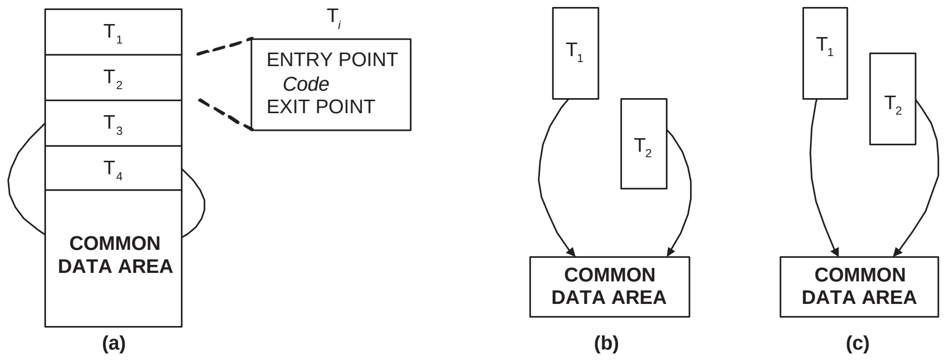

Figure 1: *Independent tasks with some common data (a), sequentially executed (b), or executed in parallel, where different tasks* ***may*** *access the same data concurrently (c).*

unpredictable behavior. A complication is that the use of synchronization is rather costly, i.e., it takes time to synchronize, and this may (in the worst case) decrease the performance of the system. A second complication is that it may be hard to find all variables that may be shared, which may come close to a re-implementation of the system by inserting synchronization primitives on every variable in the system.

The contradictory conclusion is that the shared data must be protected by synchronization, while we want to avoid this synchronization due to its cost (in time). To keep the actual number of inserted synchronizations at a minimum, we propose the use of a program analysis that can decide whether or not parallel execution of different parts of the system is safe, or if synchronization has to be inserted. The analysis should also be able to decide whether or not a given part of the system is re-entrant, meaning that the same part can be executed in several ”instances” without the need for synchronization. To ensure the correctness of the analysis, i.e., that the result of the analysis is consistent with the semantics of the implementation language, the analysis must be based on a formal semantics of the same language.

Our subject of study is the language PLEX, which is used to program the functionality in the AXE telephone exchange system from Ericsson. The decision to work with PLEX is motivated by the facts that the language (and its execution model) (1) in its structure is pseudo-parallel with independent tasks executed in a non-preemptive fashion in combination with unprotected, shared data, which implies assumptions on sequential execution, and (2) with the presence of a large amount of applications written in PLEX, as well as an experimental, shared-memory architecture, we don't have to restrict future evaluations to toy-like examples.

This thesis will present a small-steps operational semantics for PLEX, both for a single-processor architecture, as well as for a multi-processor ditto. The main motivation for the semantics, is the necessity for a future program analysis, that can classify parallel execution as safe (or unsafe), to be based on a formal semantics of the language.

The thesis will also propose ”guidelines”, in the form of informal criteria, for how the existing code, by a minimum of changes, could be transformed into suitable parallel code, something that further motivates the development of a formal semantics: since we must be able to guarantee that the proposed transformations does not introduce any errors in the system, i.e., that we don't change its behavior, our guidelines must be based on formal models of both sequential, and parallel execution. This guarantee is necessary due to the very high availability demands that exists for telephone exchange systems.

Finally, by introducing primitives for synchronization in the language, as well as specifying its semantics, the thesis will propose a new language, ”Parallel PLEX”, suitable for parallel processing.

2 Programming Language Semantics

Programming language semantics is concerned with rigorously specifying the meaning, or *the effect*, of programs that are to be executed. By effect we mean, for instance, the contents of the memory locations, which parts of the program that is to be executed, or the behavior of the hardware affected by the program. A semantic specification captures these things in a formal way, and while the standard literature [18, 22] contains several arguments for defining formal semantics for general programming languages, we claim that for languages used in applications with very high demands on availability (such as telephone exchanges) where it is important to guarantee the functionality of the system, one of the most important is that ”*the semantics can form the basis for analysis and verification*” [18].

We have chosen a small steps operational semantics (also denoted *structural operational semantics*) for the specification of PLEX, and we motivate our choice in the following way:

- Since we are modeling different run-time systems for PLEX, we were keen on being able to model *how* different statements are executed, which made operational semantics an obvious candidate in that it captures both the result of a computation, as well as how the result is produced.

- A small steps semantics, in opposite to a big step semantics, will also tell us how the *intermediate* steps of the execution is performed.

- Since PLEX is continuously updated, a second consideration was that the semantics should be able to use as a *reference manual* for future implementations.

- Other approaches (see the above references) could also be considered, but with the operational approach we could develop (what we thought) the most straight-forward notation.

3 Related Work

Since PLEX is used in the telecom domain, it is natural to first look at semantics for languages in the same domain since we believe that the requirements and assumptions are the same, but also semantics for imperative languages could be of interest since PLEX, if we look at the structure of the syntax, is an imperative language. Automatic parallelization in general can also be of interest, since a final goal is to automate the process of transforming sequential PLEX to its parallel counterpart. Finally, due to data encapsulation and information hiding, PLEX may also be seen as an early object based language, which motivates a study of automatic parallelization of such languages.

3.1 Telecom languages

The telecom domain contains a number of different programming, and specification languages, which have been formalized with different techniques.

CHILL (the CCITT High Level Language), which is an object-oriented language with support for concurrency [16, 21], was developed within a denotational framework called the Vienna Development Method (VDM) [15, 4], which is a specification method, that goes from abstract notation to formal specification.

The concurrent and functional language ERLANG, developed by Ericsson, and used to program the AXD switching system [7], has been specified by a structural operational semantics as part of a larger framework for formally reasoning about ERLANG programs [12].

Estelle, LOTOS, and SDL are specification languages proposed by, and used in, the telecom industry. The languages, covered in [1], are used to specify the behavior within, and between, different processes/ components, and they range from a graphical, flowchart based representation (SDL), to a more abstract, process algebraic style (LOTOS). The semantics of the latest version of SDL, SDL-2000, is based on abstract state machines [13], whereas the semantics for both Estelle, and LOTOS, is modeled by transition systems where the meaning is given by their computations [20, 5].

3.2 Semantics for imperative languages

To its syntax, PLEX is an imperative language with an asynchronous communication paradigm, and it also contains the GOTO statement. The GOTO may also be modeled in a continuation style semantics, which describes the effect of executing the remainder of the program [18], but since continuations is usually described as part of a denotational framework, and we have chosen an operational one (see Section 2), we have not considered this further.

Even though an asynchronous communication paradigm can be modeled by process algebra, we are not aware of any work that specifically models sequential asynchronous communication through queues, which is the paradigm used in the execution model for PLEX [10].

3.3 Automatic parallelization

The traditional area of application for an optimizing compiler (in which automatic parallelization is performed) has been scientific applications, where an increasing processor capacity has an major impact on the performance since much of the work in these applications can be done in parallel. The most important task for an optimizing compiler is to find, and resolve, different types of dependencies, mainly in loops and accesses of arrays, which means dependencies within the same programs/task as opposite to our situation where we want to resolve dependencies between different programs/tasks. The literature contains many surveys of optimizing compilers [14, 2, 3, 19]

3.4 Parallelization of OO languages

The only publication we are aware of that has managed to automatically transform a sequential OOP into a parallel equivalent **without** programmer annotations is [6]. Here, compile-time (static) analysis is combined with a run-time analysis, where an asynchronous thread of execution is started for every method invocation in a program.

4 Research Description

As we said in Section 1, the context of this work is to execute sequential, legacy, software on parallel hardware, without re-implementing the entire system. In order to specify a program analysis that can decide on whether parallel execution of such software is possible or not, we will define an operational semantics for an imperative language called PLEX, which is used in the telecom domain. The following is the approach of developing this semantics, as well as the contributions of this work.

4.1 The approach

- We have already defined an operational semantics for sequential execution of PLEX in its current single-processor architecture [8].
- As a first step towards a semantics for parallel execution, the semantics in [8] has been adapted to an experimental shared-memory

architecture [9]. At this point, the language has not been extended with any form of synchronization primitives, instead it executes under a restricted execution model, which prevents some parts of the programs to execute concurrently. This means that we model the situation in Fig. 1 (b), i.e., ”sequential execution on parallel hardware”.

- Before defining the semantics for ”parallel execution on parallel hardware”, we will perform a study of some existing PLEX-code in order to get an opinion on how well the existing code is suitable for parallel execution, and if not, what transformations that could be done.

- The final step will define the semantics for when the language is extended with primitives for synchronization, and we denote this language ”Parallel-PLEX”. In conjunction to this, we also intend to propose ”guidelines” for how to transform existing PLEX-code into ”Parallel-PLEX”.

4.2 The contributions

The main contributions of this thesis is the following:

- By using a labeled program (following the style in [17]) we show a straight-forward operational semantics for an imperative, non-toy like, language which includes the `GOTO` statement, and an asynchronous communication paradigm.

- We also give an example on that formal semantics is not only of theoretical interest, by taking operational semantics technology to industry, and points to an application of formal semantics that has considerable practical interest.

- In order to capture the differences between the possible sequential, and the possible parallel, executions, we show how to model both a sequential run-time system, as well as a parallel one. This will also provide us with the necessary theoretical ground for future criteria for correct parallel execution.

- We will also provide ”guidelines”, in the form of informal criteria, for when PLEX-code can be classified as re-entrant and by that executed in parallel.
- A study of existing PLEX-code, and possible propositions on how the existing code, by a minimum of changes, could be transformed into suitable parallel code.
- The design of a new language, ”Parallel PLEX”, suitable for parallel execution.

4.3 Published and planned papers

A first paper (in the form of an extended abstract) that summarized the first technical report [8], was presented at APPSEM'04, a workshop on applied semantics [11]. A second paper, that describes the operational semantics developed in the two reports [8, 9] is planned for the summer/autumn 2005. A third paper will describe the semantics for an extended language, a language where synchronization primitives has been added. Whether or not this third paper is written before or after the thesis is defended depends on the time it takes to develop this semantics.

5 Thesis Outline

A proposed title of the thesis is ”*An Operational Semantics for Parallel Execution of Re-entrant PLEX*”, and its table of contents would be the following:

1. Introduction
   In this section, we will introduce the problem with sequential software on parallel hardware, as well as stating the relevance of this work, and its contributions.
2. AXE and PLEX
   This section will introduce the AXE telephone exchange system, as well as the language PLEX, and it summarizes the technical report by Erikson and Lindell [10].

3. Architecture, and execution paradigms
   The shared memory architecture, and the different execution paradigms that is assumed by the parallel semantics will be considered in this section. We will also discuss the term re-entrant code in this section.

4. Existing source code
   In this section, we will summarize our studies of (some set of) existing PLEX-code, and discuss the possibilities of executing it in parallel, or if parallel execution is not suitable, possible transformations.

5. Operational semantics
   This section is intended to give the reader not familiar with semantic notation the necessary background before approaching the operational semantics for PLEX. We will also argue on why we chose an operational semantics to describe the language.

6. Core-PLEX and its operational semantics
   For readability and simplicity, we have defined a subset of PLEX denoted *Core-PLEX*, which will be defined in the beginning of this section. In the following subsections we will define the formal semantics, which is a conventional structural operational semantics in the style used in [18], for this subset.

   (a) Sequential execution on sequential hardware
       This subsection defines the operational semantics for Core-PLEX when the language executes on a single-processor architecture.

   (b) Sequential execution on parallel hardware
       This subsection defines the semantics when the language, without any synchronization, executes on an experimental shared-memory architecture. At this point, the language executes under a restricted execution model which prevents some parts of it to be executed concurrently.

   (c) Parallel execution on parallel hardware
       The last subsection defines the semantics for ”Parallel-PLEX”,

i.e., the same subset as in the previous sections, but extended with primitives for synchronization.

We also intend to discuss how PLEX, by manual changes, could be turned into its parallel counterpart, ”Parallel-PLEX”, and thereby be classified as re-entrant.

7. Related Work
   This section will give an overview of related work, mainly in the area of formal descriptions for implementation, and specification, languages used in the telecom domain, but also semantics for imperative languages that contains the `GOTO` statement and an asynchronous communication paradigm (if such work can be found). We also intend to discuss some general works in the area of (automatic) parallelization.

8. Conclusions and future work
   This section will summarize the contribution of this thesis, and also discuss the future work of the project.

6 Time-Plan

This sections contains the proposed time plan for the remaining parts of the licentiate thesis project. The sequential semantics has already been developed [8], and the semantics for parallel execution under a restricted execution model is about to be finished [9]. The licentiate degree requirement on 30 credits will be fulfilled with the course in *Program Analysis*.

2005-06-22	Licentiate proposal
2005-07-31	Course in Program Analysis finished
2005-07-31	Second technical report [9] finished
2005-08-31	Literature survey on parallelization finished
2005-10-31	Study of existing PLEX-code finished
2005-12-31	Third step of the semantics completed (including a technical report)
2006-02-15	Thesis draft ready for review
2006-03-31	Licentiate thesis finished and defended

7 Futute Work

Future work includes developing a program analysis that states if parallel execution of different PLEX programs is safe or not, or if synchronization primitives has to be inserted. The semantics developed in this thesis will provide the formal basis for this analysis in that the semantics is used to ensure that the result from the analysis is consistent with the semantics of the language.

References

[1] M. A. Ardis. Formal Methods for Telecommunication System Requirements: A Survey of Standardized Languages. *Annals of Software Engineering*, 3:157–187, 1997.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[4] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[5] M. Calder and C. Shankland. A Symbolic Semantics and Bisimulation for Full LOTOS. In *Proceedings of the 21st International Conferenence on Formal Techniquess for Networked and Distributed Systems*, pages 185–200. IFIP, 2001.

[6] B. Chan and T. S. Abdelrahman. Run-Time Support for the Automatic Parallelization of Java Programs. *The Journal of Supercomputing*, 28(1):91–117, 2004.

[7] B. Däcker. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Licentiate thesis, Royal Institute of Technology, KTH, Sweden, 2000.

[8] J. Erikson. A Structural Operational Semantics for PLEX. MRTC Report, ISSN 1404-3041 ISRN MDH-MRTC-166/2004-1-SE, Mälardalen University, 2003.

[9] J. Erikson. An Operational Semantics for the Execution of PLEX in a Shared Memory Architecture. Technical report, Mälardalen University, **To be published**, 2005.

[10] J. Erikson and B. Lindell. The Execution Model of the APZ/PLEX - An Informal Description. Technical report, Mälardalen University, 2002.

[11] J. Erikson and B. Lisper. A Formal Semantics for PLEX. In *Proceedings of the 2nd APPSEM II Workshop, APPSEM'04*, Tallin, 14-16 April 2004.

[12] L. Fredlund. *A Framework for Reasoning About ERLANG Code*. PhD thesis, Royal Institute of Technology, KTH, Sweden, 2001.

[13] U. Glässer, R. Gotzhein, and A. Prinz. The Formal Semantics of SDL-2000: Status and Perspectives. *Computer Networks - The International Journal of Computer and Telecommunications Networking*, 3(42):343–358, June 2003.

[14] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation Techniques for Parallel Systems. *Parallel Computing*, 25(13-14):1741–1783, 1999.

[15] ITU-T. *CHILL: Formal Definition*, 1982. International Telecommunication Union, Volume 1, Part 1, 2, 3.

[16] ITU-T. *CHILL: The ITU-T Programming Language*, 11 1999. International Telecommunication Union, Geneva, (Recommendation Z.200).

[17] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2nd Edition*. Springer, 2005.

[18] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[19] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[20] J. Thees and R. Gotzhein. A Formal Syntax and a Formal Semantics for Open Estelle. Technical Report 292/97, University of Kaiserslautern, 1997.

[21] J. F. H. Winkler. CHILL 2000. *Telektronikk*, 96(4):70–77, 2000.

[22] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.