Two Formal Semantics for PLEX

Johan Erikson and Björn Lisper
Dept. of Computer Science and Electronics, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN
{johan.erikson, bjorn.lisper}@mdh.se

Abstract

In any system with shared data and concurrent (or independent) activities, there is a need to guarantee exclusive access to the shared data. If the independent parts are executed in a non-preemptive fashion, on a single-processor architecture, the exclusive access is automatically guaranteed. The problem arises when the single-processor architecture is to be replaced by a multi-processor ditto; different parts, still executed in a non-preemptive fashion, but on different processors, may now access and update the same data concurrently.

Since re-implementation is an infeasible solution for a legacy system, which usually contains several million lines of code, there is a need for criteria that can ensure when functional equivalence, in some central aspect, is preserved for parallel execution of the software. To ensure the correctness of these criteria, the formal semantics of the language in question has to be considered.

The AXE telephone exchange system from Ericsson exposes the above properties: independent activities, shared data, non-preemptive execution, and a single-processor architecture. PLEX is used to program the functionality in the system, and in this paper we develop an operational semantics for the current single-processor architecture, as well as for an experimental multi-threaded, shared-memory, architecture.

1 Introduction

In a system with independent activities and shared data, the issue of exclusive access to the shared data need to be handled. If the system has been designed for a multi-processor architecture, it is probable that the shared data is protected by some form of synchronization, and the independent parts are executed concurrently. But if parallel processing, and synchronization, wasn't an issue at the time of designing the system, the different parts can be executed in a non-preemptive fashion (run to termination without interruption) which, on a single-processor architecture, automatically protects the shared data since independent activities are executed in

sequence. We denote the software in the second case as 'sequential software'.

While legacy software systems, developed and maintained over many years, contains large amounts of sequential software (executed on single-processor architectures), there is a development towards different forms of parallel hardware. The problem arises when the single-processor architecture is to be replaced by a multi-processor ditto where the independent parts are executed concurrently. At this point, the non-preemptive execution does not protect the shared data any longer, since independent parts that are executed on different processors now may access and update the same data concurrently. The question is: *How is such a system to be parallelized?*

A naive solution would be to re-design and re-implement the system, but since a legacy software system usually contains several million lines of code, this solution is infeasible. However, when data is shared, access to it must be synchronized to avoid an unpredictable behavior, but since the use of synchronization is rather costly, i.e., it takes time to synchronize, we also want to avoid this synchronization due to its cost. To keep the actual number of inserted synchronizations at a minimum, we would need criteria that could ensure when functional equivalence, in some central aspect, is preserved for the sequential software when executed on parallel hardware, and to be able to ensure the correctness of such criteria, the formal semantics of the language in question has to be considered.

Our subject of study is the language PLEX, used to program the functionality in the AXE telephone exchange system from Ericsson. The above properties: independent activities, shared data, non-preemptive execution, and a single-processor architecture are all present in the system. In this paper, we describe an operational semantics for the current single-processor architecture, as well as for an experimental multi-threaded, shared-memory, architecture. We have modeled the most important parts of PLEX in a simplified subset denoted *Core PLEX*, where some of the details in the original language and its execution model has been omitted.

The rest of this paper is organized as follows: related work is covered in Section 2, and the most important parts of PLEX in Section 3. Section 4 covers our subset of PLEX, Core PLEX, as well as the sequential semantics for the same subset, whereas Section 5 briefly describes the experimental shared-memory architecture, its execution model, and the extended semantics for Core PLEX. The work is summarized in Section 6, where we also discuss future work.

2 Related Work

Since PLEX is used in the telecom domain, we will focus on semantics for languages in the same domain since we believe that the requirements and assumptions are the same.

CHILL (the CCITT High Level Language), an object-oriented language with support for concurrency [8], is specified within a denotational framework [7]. The concurrent and functional language ERLANG, developed by Ericsson, and used to program the AXD switching system, has been specified by a structural operational semantics as part of a larger framework for formally reasoning about ERLANG programs [5]. 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 [6], whereas the semantics for both Estelle, and LOTOS, is modeled by transition systems where the meaning is given by their computations [11, 2].

3 Programming Language for EXchanges

PLEX is used to program the functionality in the AXE telephone exchange system (developed in the 1970's). Besides implementation of new functionality, there is also a large amount of existing PLEX code to maintain. Apart from an asynchronous communication paradigm, PLEX is an imperative language, with assignments, conditionals, goto's, and a restricted iteration construct (which only iterates between given start and stop values). It lacks some common statements from other programming languages such as `WHILE` loops, negative numeric values and real numbers.

A PLEX program file (called a *block*) consists of several, independent *sub-programs* together with block wise scooped data. The sub-programs can be executed in any order, and one or several sub-programs constitutes a *Job*, which is a continuous sequence of statements executed in the processor. Due to the independent sub-programs, it is more accurate to talk about the execution of a number of independent and "parallel" jobs, than of the execution of the PLEX program file. However, the jobs are not executed truly in parallel: rather, when spawned, they are buffered (queued), and sequentially executed in a non-preemptive fashion according to a FIFO-semantics, see Fig 1 (b). Because of the buffering of spawned jobs, we term the language as "pseudo-parallel".

Jobs communicate and control other jobs through a kind of events called *signals*; a job begins with a signal receiving statement and is terminated by the use of an `EXIT` statement. Signals are classified through combinations of different properties, where the main distinction, from a semantical point of view, is between *direct* and *buffered* signals, see Figure 1. The difference is that a direct signal continues an ongoing job, whereas a buffered signal spawns off a new job. A direct signal is in this way similar to a jump (with data) that retains control over the execution.

Until recently, the semantics for PLEX has been defined through its

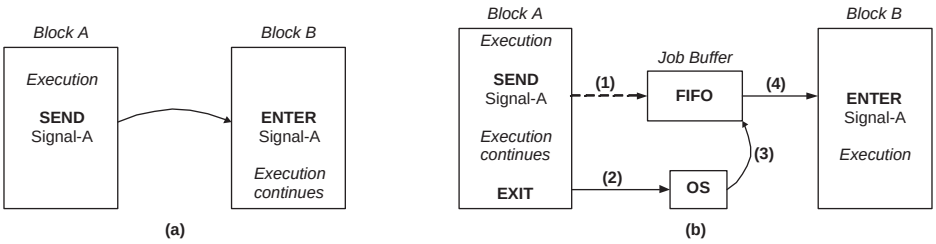

Figure 1: *(a): a direct signal, "similar" to a jump. (b):buffered signals: Block A sends a buffered signal which is inserted at the end of the job buffer (1). When Block A terminates, the control is transferred to the OS (2), which fetches a new signal from the buffer (3) (**Note**: The signal fetched at (3) does not have to be the same signal that was inserted at (1) since the buffers have a FIFO-semantics.) The signal then triggers the execution of Block B (4).*

implementation, but in previous work [4, 3], we have presented a sequential, as well as a parallel, structural operational semantics (in the style used in [10]) for a substantial subset of PLEX. The meaning of the language has thereby been given a formal specification, something that is required for formal verification of the correctness of parallel implementations, both the one modeled in this paper, and more aggressive implementations/parallelizations based on program analysis.

4 Core PLEX, and a Sequential Semantics

We begin this section by defining the subset of PLEX that we, in Section 1, denoted Core PLEX, before we discuss the sequential semantics for this subset. The extensions to cope with the experimental multi-threaded, shared-memory, architecture are discussed in Section 5. In both cases, *we assume that the modeled activities terminates*.

Besides some ordinary imperative constructs, including the `GOTO` statement, Core PLEX contains the statements: `SEND` $signal$, `RECEIVE` $signal$, and `EXIT`, since these are the statements needed to control a job (see Section 3). The syntactic categories, as well as the abstract syntax, are shown in Table 1, and as could be seen in the table, we use a labeled program (as in [9]) to identify the individual, 'atomic' statements. To determine the program label of a certain statement, we use the function $Label$, which returns the label of the given statement (or, in case of a sequence of statements, the label of the first statement in the sequence). We assume that the program is $label\ consistent$, meaning that each label occurs only once. We will also take advantage of the fact that a signal can only be received by a single (unique) receiving statement, and according to this, assume a direct mapping from the name of a signal to the label of its corresponding receiving statement.

a	$\in$	**AExp**	arithmetic expressions
b	$\in$	**BExp**	boolean expressions
S	$\in$	**Stmt**	statements
l	$\in$	**Lab**	labels
$x,\ y$	$\in$	**Var**	variables
n	$\in$	**Num**	numerals
op_a	$\in$	arithmetic operators	
op_r	$\in$	relational operators	
a	::=	$x \mid n \mid a_1\ op_a\ a_2$	
b	::=	$a_1\ op_r\ a_2$	
S	::=	$[x := a]^l \mid S_1;\ S_2 \mid [\texttt{GOTO}\ label]^l \mid [\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2]^l \mid$ $[\texttt{SEND}\ signal]^l \mid [\texttt{ENTER}\ signal]^l \mid [\texttt{EXIT}]^l$	

Table 1: *The abstract syntax for "Core PLEX".*

Since we have assumed a labeled consistent program, there is a 1:1 mapping between labels and statements, and we can assume the inverse of the $Label$ function, $Label^{-1}$, which takes a program label as input and returns the corresponding statement.

The execution of statements is modeled by state transitions, where the state that we model is determined by the current implementation of PLEX, as well as by the underlying architecture/execution model, but with some of the details in the original language, and the execution model, omitted (as said in Section 1). Since the language allows different kinds of jumps, we have introduced a *virtual statement counter*, $\mathcal{VSC}$, which maps to the program label of the statement to be executed, and for convenience, we use $\mathcal{VSC}$++ to denote $\mathcal{VSC}$ = $\mathcal{VSC}$+1. Apart from this counter, the state is defined by the contents in the memory, and the job buffer. A state s is thus a tuple:

$$s := \langle \mathcal{VSC}, RM, DS, JBB \rangle$$

where Table 2 explains each component of the state. Now, the transition rules, defining the semantics, are quite straightforward. Assignments will affect the proper memory component of the state; explicit jumps the virtual statement counter. Compound statements (conditionals, and sequenced statements) have standard transition rules defined inductively over the structure of the statement. The resulting semantics is summarized in Table 4, and to denote the value of any of the components in s (for instance the $\mathcal{VSC}$) in a new state s', we have used the notation $\mathcal{VSC}'$ which is to be interpreted as $s'(\mathcal{VSC})$. In order to manipulate the job buffer, which is modeled as a list of signals, we use the notation $signal : JBB$ to denote a list with the head $signal$, and the tail JBB, and $JBB : signal$ to denote that we append $signal$ to the list JBB.

The functions that are used both in Table 4 and Table 6 (or only in Table

4) are defined in Table 3. The functions $\mathcal{A}$ and $\mathcal{B}$ have standard definitions, i.e., to evaluate an arithmetic, and a boolean expression respectively, and are omitted in the table.

Item	**Explanation**	**Set of values**
$\mathcal{VSC}$	*Virtual Statement Counter*	**labels**
RM	*Register Memory (Temporary data)*	$x \rightarrow \mathbf{N}$
DS	*Data Store (Shared data)*	$x \rightarrow \mathbf{N}$
JBB	*Job Buffer*	$list\ of\ signals$

Table 2: *The different items in the (single-pro) state of the system.*

$$\begin{array}{l}\mathcal{P} : \mathbf{Var} \rightarrow \{\texttt{RM, DS}\} \\ \mathcal{SD} : signal \rightarrow \{\texttt{DIRECT, BUFFER}\} \\ Label : \mathbf{Stmt} \rightarrow \mathbf{Lab} \\ \mathcal{APZ} : state \hookrightarrow signal\end{array}$$

Table 3: *Functions that are used both in the sequential (Table 4), and the parallel (Table 6) semantics. $\mathcal{APZ}$, which is a partial function due to the possibility of an empty job buffer, is re-defined in Table 5.*

5 Shared Memory, and a Parallel Semantics

In Section 4, we said that the state that was modeled by the sequential semantics was determined by the current implementation of PLEX and of the underlying execution model. This holds for the parallel semantics as well, where the new state, with k concurrently executing threads, is determined by an experimental multi-threaded, shared-memory, architecture. The new state s contains one part, s_G, that can be modified by **any** of the k threads (including the shared data, DS), and k local parts (s_i) that are local to thread T_i (including the storage for temporary data, RM_i):

$$s := \langle s_0, \ldots, s_k, s_G \rangle \text{ where } s_i := \langle \mathcal{VSC}_i, RM_i, JBB_i, Locks_i \rangle, \quad 0 \leq i \leq k$$

$$\text{and } s_G := \langle DS, L_0, \ldots, L_\beta \rangle \text{ where } \beta \in \{\mathbf{N}\}$$

The architecture, and its execution model, is designed to be 'functionally equivalent' with the single-processor system. This is achieved by restricting parallel execution to jobs from different *Job-Trees* (informally defined as the set of jobs originating from the same external signal), whereas jobs from the same Job-Tree executes in the same sequential order as in the single-processor architecture. Exclusive access to the shared data is ensured by the run-time system, which utilizes a number of binary locks, $L_0, \ldots, L_\beta$,

[ass^{RM}]	$\langle [x := a]^l, s\rangle \;\Rightarrow\; s[\mathcal{VSC}{+}{+},\ RM(x) \mapsto \mathcal{A}[\![a]\!]s]$	**if** $\mathcal{P}(x) = RM$
[ass^{DS}]	$\langle [x := a]^l, s\rangle \;\Rightarrow\; s[\mathcal{VSC}{+}{+},\ DS(x) \mapsto \mathcal{A}[\![a]\!]s]$	**if** $\mathcal{P}(x) = DS$
[seq]	$\langle S_1;\ S_2,\ s\rangle \Rightarrow \langle S_1';\ S_2,\ s'\rangle$	**if** $\langle S,\ s\rangle \Rightarrow \langle S',\ s'\rangle$
[seq]	$\langle S_1;\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle$	**if** $\langle S,\ s\rangle \Rightarrow s'$, **and** $\mathcal{VSC}' = Label(S_2)$
[jmp]	$\langle [\texttt{GOTO}\ label]^l, s\rangle \;\Rightarrow\; \langle S,\ s[\mathcal{VSC} \mapsto label]\rangle$	**if** $S = Label^{-1}(label)$
[cond^{tt}]	$\langle [\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2]^l,\ s\rangle \;\Rightarrow\; \langle S_1,\ s\rangle$	**if** $\mathcal{B}[\![b]\!]s = \texttt{tt}$
[cond^{ff}]	$\langle [\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2]^l,\ s\rangle \;\Rightarrow\; \langle S_2,\ s\rangle$	**if** $\mathcal{B}[\![b]\!]s = \texttt{ff}$
[send^{dir}]	$\langle [\texttt{SEND}\ signal]^l,\ s\rangle \Rightarrow$ $\langle S,\ s[\mathcal{VSC} \mapsto signal,\ RM \mapsto \texttt{UNDEF}]\rangle$	**if** $\mathcal{SD}(signal)$ $=$ DIRECT**, and** $S = Label^{-1}(signal)$
[send^{buf}]	$\langle [\texttt{SEND}\ signal]^l,\ s\rangle \Rightarrow$ $\langle S,\ s[\mathcal{VSC}{+}{+},\ JBB : signal]\rangle$ **where** $s = \langle \ldots,\ JBB,\ \ldots\rangle$	**if** $\mathcal{SD}(signal)$ $=$ BUFFER, **and** $S = Label^{-1}(\mathcal{VSC}{+}{+})$
[enter^{dir}]	$\langle [\texttt{ENTER}\ signal]^l,\ s\rangle \Rightarrow$ $\langle S,\ s[\mathcal{VSC}{+}{+},\ RM \mapsto \texttt{UNDEF}]\rangle$	**if** $\mathcal{SD}(signal)$ $=$ DIRECT**, and** $S = Label^{-1}(\mathcal{VSC}{+}{+})$
[enter^{buf}]	$\langle [\texttt{ENTER}\ signal]^l,\ s\rangle \Rightarrow$ $\langle S,\ s[\mathcal{VSC}{+}{+},\ JBB,\ RM \mapsto \texttt{UNDEF}]\rangle$ **where** $s = \langle \ldots,\ signal : JBB,\ \ldots\rangle$	**if** $\mathcal{SD}(signal)$ $=$ BUFFER, **and** $S = Label^{-1}(\mathcal{VSC}{+}{+})$
[exit]	$\langle [\texttt{EXIT}]^l,\ s\rangle \Rightarrow$ $\langle S,\ s[\mathcal{VSC} \mapsto \mathcal{APZ}(s),\ RM \mapsto \texttt{UNDEF}]\rangle$	**if** $S =$ $Label^{-1}(\mathcal{APZ}(s))$

Table 4: *The operational semantics for sequential "Core PLEX"*

.

to prevent simultaneous access to the same block. $\mathcal{BLK}$ returns the lock associated with a given block, and the number of locks that a job "collects" during its execution, are collected in the set $Locks_i$. Possible deadlock situations are resolved by the run-time system. Without any formal proof, we intuitively say that the parallel semantics, from a Job-Tree point of view, is equivalent with the sequential semantics based on the following observations: (1) jobs in the same Job-Tree are executed in the same sequential order in both cases, (2) a job is still ensured exclusive access to the shared data in a block since the run-time system prevents other jobs from simultaneous access to the block (and its data), and (3) since external signals (from which a Job-Tree originates) may arrive in arbitrary order, the Job-Trees may be executed in different orders both in the sequential, as well as in the parallel case.

To distinguish between the threads when we look at their concurrent execution, we change the configurations from $\langle S,\ s\rangle$ to $\langle S,\ i,\ s\rangle$ to capture the execution of S by thread T_i. The resulting semantics is shown in Table 6, whereas Table 5 summarizes the functions introduced in Table 6.

$$\mathcal{APZ} : \{0, 1,\ \ldots, k\} \times state \hookrightarrow state$$
$$\mathcal{BLK} : state \times signal \rightarrow \mathbf{Blk}$$

Table 5: *The functions that are introduced in (Table 6).*

To denote a state with k numbers of concurrently executing threads, we use the notation S_i to denote the execution of statement S by thread T_i, which together with | as our parallelizing operator, leads to the following description of a running system (with k concurrently executing threads, and a state s): $\langle S_0 \mid S_1 \mid \ldots \mid S_k,\ s\rangle$ and we say that there will be a change in the global state of the form $\langle \ldots \mid S_i \mid \ldots,\ s\rangle \Rightarrow \langle \ldots \mid S_i' \mid \ldots,\ s'\rangle$ if there is a valid, local transition of the form $\langle S,\ i,\ s\rangle \Rightarrow s'$, or $\langle S,\ i,\ s\rangle \Rightarrow \langle S',\ i,\ s'\rangle$, which leads to the last two rules in Table 6, valid for any i.

6 Summary

This paper presents two operational semantics for Core PLEX, a subset of PLEX, used to program functionality in the AXE system; one that models execution on the current single-processor architecture, and one for an experimental multi-threaded, shared-memory, architecture. The parallel semantics models a restricted execution model where related activities are prevented from being executed in parallel. A more aggressive parallelization would allow these activities to execute in parallel, but parallel execution also means that the language has to be extended with primitives for synchronization to protect the shared data. To keep the actual number of inserted synchronizations at a minimum, we need criteria that ensures when parallel execution of the current software is safe in the sense that

[ass^{RM}]	$\langle [x := a]^l,\ i, s\rangle \ \Rightarrow\ s[\mathcal{VSC}_i{+}{+},\ RM_i(x) \mapsto \mathcal{A}[\![a]\!]s]$	**if** $\mathcal{P}(x) = RM$
[ass^{DS}]	$\langle [x := a]^l,\ i, s\rangle \ \Rightarrow\ s[\mathcal{VSC}_i{+}{+},\ DS(x) \mapsto \mathcal{A}[\![a]\!]s]$	**if** $\mathcal{P}(x) = DS$
[seq]	$\langle S_1;\ S_2,\ i, s\rangle \ \Rightarrow\ \langle S_1';\ S_2,\ i, s'\rangle$	**if** $\langle S,\ i,\ s\rangle \Rightarrow \langle S',\ i,\ s'\rangle$
[seq]	$\langle S_1;\ S_2,\ i, s\rangle \ \Rightarrow\ \langle S_2,\ i, s'\rangle$	**if** $\langle S,\ i,\ s\rangle \Rightarrow s'$, **and** $\mathcal{VSC}_i' = Label(S_2)$
[jmp]	$\langle [\texttt{GOTO}\ label]^l,\ i, s\rangle \ \Rightarrow\ \langle S,\ i,\ s[\mathcal{VSC}_i \mapsto label]\rangle$	**if** $S = Label^{-1}(label)$
[cond^{tt}]	$\langle [\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2]^l,\ i, s\rangle \ \Rightarrow\ \langle S_1,\ i, s\rangle$	**if** $\mathcal{B}[\![b]\!]s = \texttt{tt}$
[cond^{ff}]	$\langle [\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2]^l,\ i, s\rangle \ \Rightarrow\ \langle S_2,\ i, s\rangle$	**if** $\mathcal{B}[\![b]\!]s = \texttt{ff}$
[send^{dir}]	$\langle [\texttt{SEND}\ signal]^l,\ i, s\rangle \ \Rightarrow$ $\langle S,\ i,\ s[\mathcal{VSC}_i \mapsto signal,\ RM_i \mapsto \texttt{UNDEF},$ $Locks_i\ :=\ Locks_i\ \cup\ \{L_\gamma\},\ L_\gamma \mapsto 1]\rangle$ **where** $L_\gamma\ =\ \mathcal{BLK}(s,\ signal)$	**if** $\mathcal{SD}(signal)$ $=$ DIRECT, $L_\gamma\ =\ 0$, **and** $S = Label^{-1}(signal)$
[send^{buf}]	$\langle [\texttt{SEND}\ signal]^l,\ i, s\rangle \ \Rightarrow$ $\langle S,\ i,\ s[\mathcal{VSC}_i{+}{+},\ JBB_i : signal]\rangle$ **where** $s = \langle \ldots,\ JBB,\ \ldots\rangle$	**if** $\mathcal{SD}(signal)$ $=$ BUFFER, **and** $S = Label^{-1}(\mathcal{VSC}_i{+}{+})$
[enter^{dir}]	$\langle [\texttt{ENTER}\ signal]^l,\ i, s\rangle \ \Rightarrow$ $\langle S,\ i,\ s[\mathcal{VSC}_i{+}{+},\ RM_i \mapsto \texttt{UNDEF}]\rangle$	**if** $\mathcal{SD}(signal)$ $=$ DIRECT, **and** $S = Label^{-1}(\mathcal{VSC}_i{+}{+})$
[enter^{buf}]	$\langle [\texttt{ENTER}\ signal]^l,\ i,\ s\rangle \ \Rightarrow$ $\langle S,\ i,\ s[\mathcal{VSC}_i{+}{+},\ JBB_i,\ Locks_i\ :=$ $Locks_i\ \cup\ \{L_\gamma\},\ RM_i \mapsto \texttt{UNDEF}]\rangle$ **where** $s = \langle \ldots,\ signal : JBB_i,\ \ldots\rangle$	**if** $\mathcal{SD}(signal)$ $=$ BUFFER, **and** $S = Label^{-1}(\mathcal{VSC}_i{+}{+})$
[exit]	$\langle [\texttt{EXIT}]^l,\ i, s\rangle \ \Rightarrow\ \langle S,\ i,\ s'\rangle$	**if** $s' = \mathcal{APZ}(i, s)$, **and**, $S = Label^{-1}(\mathcal{VSC}_i')$
[par]	$\dfrac{\langle S,\ i,\ s\rangle \ \Rightarrow\ s'}{\langle \ldots \mid S_i \mid \ldots,\ s\rangle \ \Rightarrow\ \langle \ldots \mid S_i' \mid \ldots,\ s'\rangle}$	**if** $S_i' = Label^{-1}(\mathcal{VSC}_i')$
[par]	$\dfrac{\langle S,\ i,\ s\rangle \ \Rightarrow\ \langle S',\ i,\ s'\rangle}{\langle \ldots \mid S_i \mid \ldots,\ s\rangle \ \Rightarrow\ \langle \ldots \mid S_i' \mid \ldots,\ s'\rangle}$	

Table 6: *The operational semantics for the multi-threaded Core PLEX.*

functional equivalence is preserved. To ensure the correctness of such criteria, the formal semantics of the language has to be considered. Future work will add primitives for synchronizations to PLEX, as well as specifying the semantics for this extended language. We will also formally define 'functional equivalence', and prove that the property holds between the different semantics.

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] 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.

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

[4] 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.

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

[6] 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.

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

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

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

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

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