A Formal Semantics for PLEX

Johan Erikson and Björn Lisper
Dept. of Computer Science and Engineering, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN

PLEX (Programming Language for EXchanges) is a special-purpose, pseudo-parallel, event-based real-time language developed by Ericsson. The language is designed exclusively for telephony systems and is used in central parts of the AXE telephone switches. It has been continuously evolving since the 1970's when it was originally designed.

PLEX has a fairly peculiar execution model. The execution of a PLEX program consists of the execution of number of parallel jobs. However, the jobs are not executed truly in parallel: rather, when spawned, they are put in one of four queues, of different priority, and sequentially executed in a non-preemptive fashion. This execution model, somewhat resembling coroutining, was probably apt for the limited hardware of the 1970's but has now turned into a problem since it is different from modern execution models. However, PLEX is not easily done away with since there are 10 million lines of code in the AXE system.

Jobs communicate and control other jobs through a kind of events called *signals*. They can be either *direct*, or *buffered*. A direct signal can be seen as a jump, whereas a buffered signal spawns a new job which is queued. A second distinction is between *single* and *combined* signals. A combined signal starts an activity which returns to the signal sending point when finished: it could thus be seen as a method or subroutine call. A single signal does not yield a return, and is thus (if direct) similar to a `GOTO` statement. An `EXIT` statement terminates a job.

Apart from the above, PLEX is an imperative language, with assignments, conditionals, goto's, and a restricted iteration construct. Scopewise, a PLEX program is organized in a flat structure of blocks containing code and data areas. These blocks can be thought of as objects, and therefore PLEX may also be seen as an early object-based language.

Hitherto, the semantics for PLEX has been defined through the single implementation that exists. There are several reasons to define a formal semantics for the language instead: it can serve as an exact documentation, which is very useful when updating the implementation, and the very high availability demands on telephone exchanges also motivates a rigorous approach to software development and maintenance necessitating a formal semantics. The biggest reason is however that Ericsson wants to introduce parallel processing in the central processor of the AXE exchange. This means that the software must be executed in parallel. PLEX programs could naturally be parallelized by running the different jobs in parallel: however, this is not always possible due to unintentional interaction between jobs through the shared data areas. To devise trustworthy criteria and program analyses for ensuring correct parallel execution requires formal semantics for both sequential and parallel execution of programs in the language.

We have defined a formal, sequential semantics for a substantial subset of PLEX. This is a quite conventional structural operational semantics in the style used in [2], where the execution of statements in the language is modeled by state transitions. However, since the language allows different kinds of unstructured jumps, we introduce a *virtual statement counter*, $\mathcal{VSC}$, which identifies the current statement to be executed. This counter is made explicit in the program state.

Apart from this counter, the state is defined by the contents in the memory, including different data areas and job queues. A state is thus a tuple:

$$\langle \mathcal{VSC}, RM, DS, RS, JBA, JBB, JBC, JBD \rangle$$

Table 1 explains each component of the state.

The transition rules, defining the semantics, are now quite straightforward to define. Assignments will affect the proper memory store component of the state; explicit jumps the virtual statement counter; indirect signals and `EXIT` statements the job buffers. Compound statements (conditionals, iteration, sequenced statements) have standard transition rules defined inductively over the structure of the statement: however, some care has to be taken to account for the possibility of unstructured jumps out of the statements. (Our semantics assumes that no unstructured jumps are made *into* structured statements: indeed, code containing such jumps is considered not well-formed in PLEX.)

Item	Explanation	Set of values
$\mathcal{VSC}$	Virtual Statement Counter	**N**
RM	Register Memory	$\mathbf{N} \cup string$
DS	Data Store	$\mathbf{N} \cup string$
RS	Reference Store	$\mathbf{N} \times \mathbf{N}$
JBA	Job Buffer - Priority A	$signal$
JBB	Job Buffer - Priority B	$signal$
JBC	Job Buffer - Priority C	$signal$
JBD	Job Buffer - Priority D	$signal$

Table 1: *Definitions of the set of values for each item in our state of the system.*

A full account for our formal PLEX semantics, including a thorough introduction to PLEX and the software system of AXE, is found in [1].

Future work includes a semantics for true parallel execution of PLEX code, which allows preemptive (i.e., interleaved) execution of concurrent jobs. Given such a semantics, it will be possible to state formal theorems about when parallel and sequential execution of PLEX programs have semantics that agree. Such theorems can then form the basis for criteria ensuring the correctness of parallel execution, as well as for program analyses identifying such situations automatically.

References

[1] J. Erikson. A Structural Operational Semantics for PLEX. Technical report, Mälardalen University, 2003.

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