An Operational Semantics for the Execution of PLEX in a Shared Memory Architecture

Johan Lindhult
Dept. of Computer Science and Electronics, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN
johan.lindhult@mdh.se

April 15, 2008

Abstract

Programming Language for EXchanges, PLEX, is a pseudo-parallel and event-driven real-time language developed by Ericsson. It is designed for, and used in, central parts of the AXE telephone switching system. The language has a signal paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. Due to the fact that a PLEX program file consists of several independent sub-programs, in combination with an execution model where new jobs are spawned and put in queues, we also classify the language as pseudo-parallel.

In previous works, we have presented a *structural operational semantics* for sequential execution of PLEX in the current single-processor architecture, i.e., a specification/formalization of the behavior of PLEX in the system as it is today.

In this report, we extend our previous work by specifying the semantics for a restricted parallel implementation of the language.

Contents

1 Introduction

A large class of computer systems has been designed under the assumption that activities in the system are executed non-preemptively (run to termination without interruption). Examples of such systems are small embedded systems that are quite static to their 'nature', or priority-based systems where activities on the highest priority are assumed to be non-interruptible. Although the reasons for non-preemptive execution may vary, common for the kind of systems we target are the usage of single-processor architectures; the combination of non-preemptive execution and a single-processor architecture automatically protects any shared data in the system, and time-consuming synchronization can be avoided. However, new architectures will increasingly be parallel [SL05]. On a parallel architecture, activities executed on different processors may access and update the same data concurrently, and non-preemptive execution does not protect the shared data any longer. On the other hand, the very idea of parallel architectures is to increase performance by parallel execution. The question is: *how utilize the power of a parallel processor for a system designed for non-preemptive execution?* A brute force solution is to re-design and re-implement the system, but especially for legacy systems (which may consist of several million lines of code) this solution is often infeasible. Automatic (or semi-automatic) methods for porting the software are therefore highly desired. However, the correctness of these methods must be ensured, especially if system failures are costly. To prove the correctness, formal semantics of the language in question has to be considered.

Our subject of study is the language PLEX, used to program the AXE telephone exchange system from Ericsson. The AXE system, and the PLEX language, developed in conjunction, have roots that go back to the late 1970's. The language is event-based in the sense that only events, encoded as signals, can trigger code execution. Signals trigger independent activities (denoted jobs), which may access shared data stored in different shared data areas. Jobs are executed in a priority-based, non-interruptable (at the same priority level), fashion on a single-processor architecture, and the language lacks constructs for synchronization.

Until recently, the semantics for PLEX has been defined through its implementation, but in previous works [Eri03, EL04], we have pre-

sented a *structural operational semantics* (in the style used in [NN92]) for sequential execution of PLEX in the current single-processor architecture. Currently, Ericsson experiments with a shared-memory architecture equipped with a run-time system that automatically protects the shared data though a locking scheme. The parallel semantics in this report models the execution of PLEX on this architecture. The semantics, which extends our previous work, is given in terms of state transitions.

The experimental architecture (which in the remaining of this report will be called the ”prototype”) is designed to be ”functionally equivalent” with the single-processor system, and it executes the existing (sequential) software without modifications (i.e., without the addition of primitives for synchronization). The approach taken to guarantee this 'equivalence' is a restricted execution model, which prevents some parts of the programs to be executed concurrently.

A more aggressive parallelization would release the above restrictions and allow several threads to execute multiple instances of the same code. But parallel execution also means that the language, most likely, has to be extended with primitives for synchronization to protect shared data. The problem is the large amount of existing PLEX code (approximately 20Mlines) which prevents re-writing of the entire system. 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 functional equivalence is preserved. To ensure the correctness of such criteria, and of the program analysis from which the criteria will be derived, the formal semantics of this extended languages has to be considered.

The rest of this report is structured in the following way: related work is covered in the following section (Section 2), and a brief introduction to the AXE telephone exchange system, as well as to the language PLEX is given in Section 3. Section 4 describes our target architecture, and Section 5 explains the different execution paradigms as well as some additional concepts that will be of importance in Section 6, in which the parallel semantics is presented. The work is summarized in Section 7.

2 Related Work

PLEX is used in the telecom domain, which has particular demands (concurrency, extreme reliability and availability, soft real-time requirements, etc.). In this domain a number of specialised programming and specification languages are used, which have been formalized with different techniques. We will mainly relate to these here.

CHILL (the CCITT High Level Language) is an object-oriented language with support for concurrency [IT99, Win00]. It was developed within a denotational framework called the Vienna Development Method (VDM) [IT82, BJ82], 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 [Däc00], has been specified by a structural operational semantics as part of a larger framework for formal reasoning about ERLANG programs [Fre01]. ERLANG is parallel by nature, and an experimental, multithreaded ERLANG implementation exists on which ERLANG programs can be directly executed without any modification [Hed98].

Estelle, LOTOS, and SDL are specification languages proposed by, and used in, the telecom industry [Ard97]. The languages are used to specify the behavior within, and between, different processes/components, and they range from a graphical, flow chart-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 [GGP03], whereas the semantics for both Estelle, and LOTOS, is modeled by transition systems where the meaning is given by their computations [TG97, CS01].

PLEX is an event-based asynchronous language. There are several event-based languages with a synchronous communication paradigm, like SIGNAL [BGJ91]. However, their synchronous nature make them guite different from PLEX, they are in general more declarative in nature, and their existing semantics have a quite different style.

PLEX has unstructured jumps. This makes it awkward to define a structural operational semantics for PLEX, and compositional reasoning becomes harder. However, Saabas and Uustalu [SU05] have recently presented a compositional, natural semantics for a language with

jumps. This kind of semantics could probably be used for PLEX as well.

3 AXE and PLEX

In this section we will only give a brief description of the AXE telephone exchange system, the language PLEX, and application modules (AM's). For a more thorough description, we refer to [EL02].

3.1 The AXE Telephone Exchange System

The AXE system, developed in its earliest version in the beginning of the 1970's, is structured in a modular and hierarchical way. It consists of the two main parts: **APT** and **APZ**, where the former is the telephony (or switching) part, and the latter is the control part. The original structure of the system (main parts of it) is shown in Fig 1. Somewhere around 1994-95, the concept of *Application Modularity* (AM) was integrated into the system. This is discussed in Section 3.3

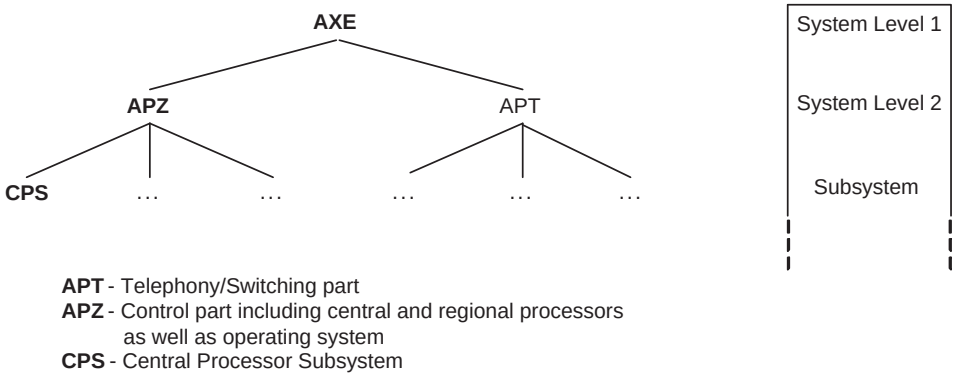

Figure 1: *The (original) hierarchical structure of the AXE system.*

The part of the system that is in focus for parallel processing is the Central Processor Sub-system, which architecture is shown in Fig. 2. In the current architecture[1], the Central Processor Sub-system consists of a *Central Processor* (CP) (which in turn consists of **a single** *CPU* and additional software), and a number of *Regional Processors* (RP's). Call

[1] Here, the "current architecture" denotes the current single-processor architecture, and **not** the parallel "prototype" that will be in focus in the remaining of this report!

requests are received by the RP's, and processed by the CP. The task of an RP, and of the CP, is described as:

Regional Processor (RP): The main task of a regional processor is to relieve the central processor by handling small routine jobs like scanning and filtering.

Central Processor (CP): This is the central control unit of the system. All complex and non-trivial decisions (such as call processing) are taken in the central processor. This is the place for all forms of non-routine work.

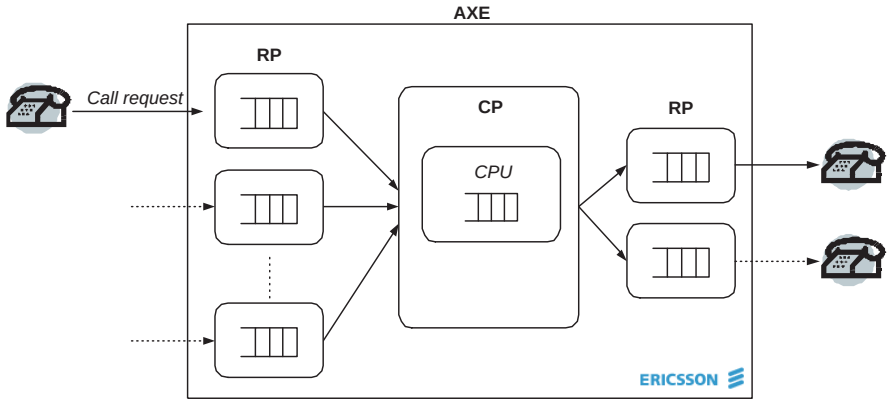

Figure 2: *Current (single-processor) architecture of the Central Processor Sub-system.*

3.2 PLEX: Programming Language for EXchanges

Programming Language for EXchanges, PLEX, is a pseudo-parallel and event-driven real-time language developed by Ericsson in conjunction with the first AXE version in the 1970's. The language is used to program the functionality in the Central Processor Sub-system, and besides implementation of new functionality, there is also a large amount of existing PLEX code to maintain. The language has a signal paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. A typical event is an incoming *call request*, see Fig. 2. Apart from an asynchronous com-

munication 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 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 scoped data, see Fig. 3. The sub-programs can be executed in any order, and one or several sub-programs constitutes a *Job* (which will be further discussed in Section 3.5). 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 non-preemptively executed in FIFO order, see Figs. 6 (b) and 4 (a). Because of the sequential FIFO order imposed, we term the language as "pseudo-parallel" since externally triggered jobs could be processed in any order (due to the order of the external signals). We also note that different types of jobs are buffered, and executed, on different levels of priority, and that jobs of the same priority are executed non-preemptively[2]. User jobs (or call processing jobs), i.e., handling of telephone calls, are always executed with high priority, whereas administrative jobs (e.g„ charging) always are executed with low priority (and never when there are user jobs to execute).

Blocks can be thought of as objects, and the subprograms are somewhat reminiscent of methods. However, there is no class system in PLEX, and it is more appropriate to view a block as a kind of software component whose interface is provided by the entry points to its sub-programs. Data within blocks is strictly hidden, and there is no other way to access it than through the sub-programs.

3.3 Application Modules

The AXE *Source System* is a number of hardware **and** software resources developed to perform specific functions according to the cus-

[2]Since jobs on the same priority level executes in a non-preemptive fashion, there are *programmer guidelines* that ought to be followed to prevent a job from executing an un proportional amount of time.

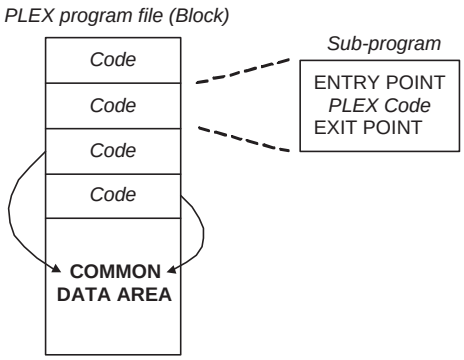

Figure 3: *A PLEX program file, called a* block*, consisting of several sub-programs.*

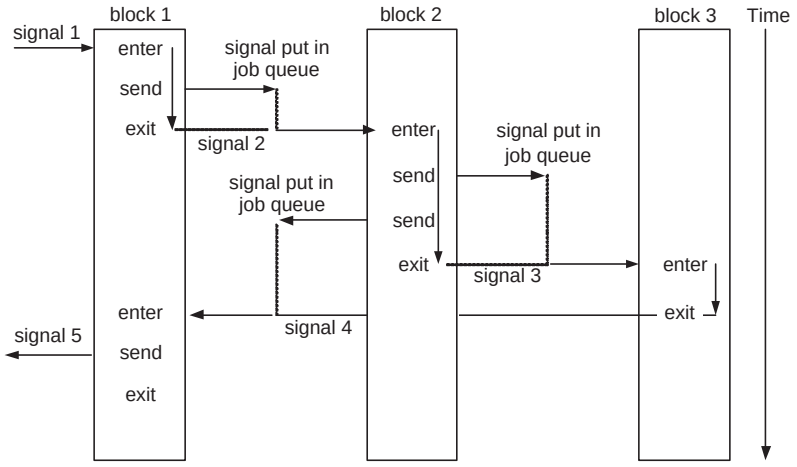

Figure 4: *The "pseudo-parallel" execution model of PLEX.*

tomer's requirements. It can be thought of as a "basket" containing all the functionality available in the AXE system. Over the years, new source systems has been developed by adding, updating or deleting functions in the original source system. But in the 1980's, the development of the AXE system for different markets (US, UK, Sweden, Asia, etc.) has led to parallel development of the source system since functionality could not easily be ported between different markets.

The solution to this increasing divergence was the *Application Modularity* (AM) concept, which made fast adaption to customer requirements possible. The AM concept specifically targeted the following requirements:

- the ability to freely combine applications in the system,
- quick implementation of requirements, and
- the reuse of existing equipment.

The basic idea is to gather related pieces of software into something called Application Modules (AMs). Different telecom applications, such as ISDN, PSTN (fixed telephony), and PLMN (Public Land Mobile Network), are then constructed by combining the necessary AMs. The idea is described in Fig. 5, where it is also shown that different AMs can be used in more than one application. The related pieces of software, mentioned above, is the PLEX blocks (Section 3.2), which means that an AM is constructed by combining the appropriate PLEX blocks, and the application by combining the appropriate AMs.

The introduction of the AM concept ended the problem with parallel development of different source systems. Instead, with AMs as building blocks, the required exchange was constructed by combining the necessary AMs into an exchange with the required functionality (i.e., with the necessary applications).

3.4 Signals

In Section 1 we said that PLEX is event-based in the sense that only events, encoded as signals, can trigger code execution. This encoding of events as signals motivates our statement that PLEX has a signal

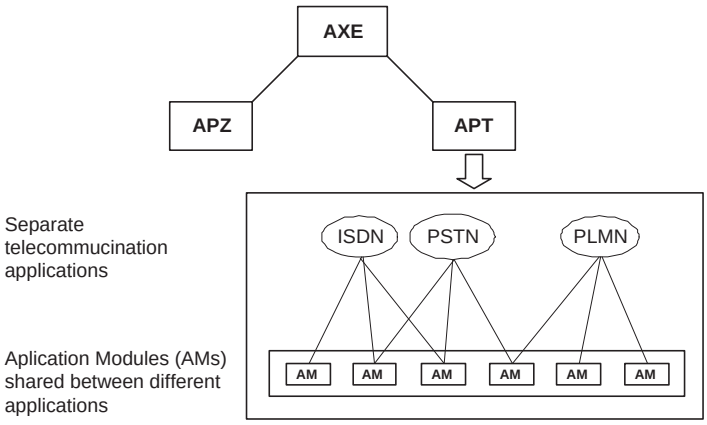

Figure 5: *The AM concept incorporated into the AXE system*.

paradigm as its top execution level. Signals are externally defined language elements for communication between different software units, and as we said in Section 3.2 this communication is asynchronous.

Every signal that is sent in the system is assigned a priority level, which is of importance when the signal is to be buffered, and it tells the ”importance” of the source code that is triggered to execution by the signal.

Signals are classified/characterized through combinations of different properties, and from a semantical point of view, the main distinction is between *direct* and *buffered* signals, Fig. 6. The difference is that a direct signal continues **an ongoing job** (discussed in Section 3.5), whereas a buffered signal spawns off **a new job**. A direct signal is in this way similar to a jump (e.g. GOTO), and by using direct signals, the programmer retains control over the execution. However, direct signals are normally only allowed to be used in very time-critical program sequences, such as call set-up routines. Buffered signals, on the other hand, are put in special (FIFO-)queues (called job buffers) when they are sent from a job, and when that job terminates, the operating system will fetch the first inserted buffered signal and start a new job, see Fig. 6. This means that after the sending of the buffered signal, the two, resulting ”execution paths” are independent of each other, but there may still be a ”sequencing issue”, though, as the jobs have to execute in the order imposed by the corresponding $Job\text{-}Tree$. We informally define a

Job-Tree as the set of jobs originating from the same external signal, and the subject will be further discussed in Section 5.1.1.

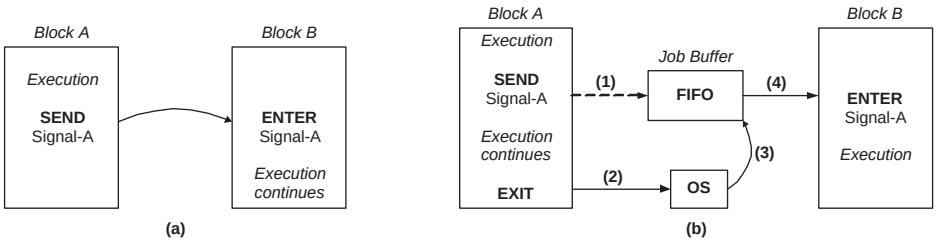

Figure 6: *(a): a direct signal, "similar" to a jump. (b):buffered signals: Block A sends a buffered signal which is inserted at the end of the job buffer (1). When Block A terminates, the control is transferred to the OS (2), which fetches a new signal from the buffer (3) (**Note**: The signal fetched 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).*

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, see Fig. 7. Note that a combined signal is always direct, while the single signal may be buffered.

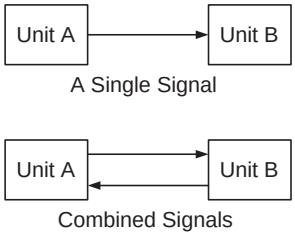

Figure 7: *Single and combined signals.*

Third, we also distinguish between *external*, and *internal* signals, where the latter is issued from an ongoing job by a `SEND` statement. External signals, on the other hand, are the signals that are sent from an RP to the CP (e.g., as a result of a call request), see Fig. 2.

A final distinction can also be made between *local* and *non-local* signals, where the former is a signal that is sent between sub-programs in the same block, and the latter between sub-programs in different blocks.

3.5 Jobs

In Section 3.2, we said that 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, and in the preceding section we have seen how jobs communicate and control other jobs through a kind of events called signals. But what is a *Job* ?

A job is a continuous sequence of statements executed in the processor, starting with the execution of an `ENTER` statement for a buffered signal and is terminated by the execution of an `EXIT` statement, and we say that a job have a *Single-Entry-Multiple-Exit* semantics, since it **always** have a single entry point but it *may* have multiple exit points.

An `ENTER` statement for a direct signal does **not** start a new job, instead it continues an ongoing job.

A job is not limited to one sub-program, several sub-programs (in different blocks) may form a job.

4 The Architecture

As we said in Section 1, the parallel semantics in this report models the execution of PLEX on an experimental parallel architecture, but before we discuss the underlying execution model that is modeled by the semantics (which will be done in Section 5.4), we need to say something about the parallel architecture.

The execution paradigms that are considered in Section 5.2 - 5.4, all requires a shared memory architecture with support for Thread-Level Parallelism (TLP). Examples of such architectures are Symmetric Multiprocessors (SMP), Chip-Multiprocessors (CMP), and Simultaneous Multi-Threading processors (SMT), which means that this (i.e., the shared memory architecture in Fig. 8) will be the architecture assumed by the semantics in Section 6. The parallel prototype uses a locking scheme to protect a block from being concurrently accessed by two different jobs. This introduces the risk of deadlocks. However, the

prototype has a mechanism to resolve this at runtime, and we will not consider this further in this report!

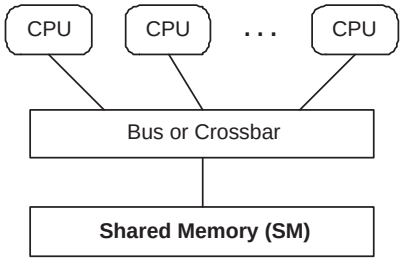

Figure 8: *The shared memory, multi-processor, architecture that is considered in this report.*

5 Execution Paradigms

Before we explain the different execution models (Section 5.2 - 5.4) that has been, and are, considered by Ericsson as possible execution paradigms for parallel execution of PLEX, we will introduce some additional concepts (Section 5.1) that are not found in [EL02, Eri03], and which are of importance in Section 6 where we present the parallel semantics for PLEX.

The material in Sections 5.2 - 5.4, are mainly collected from [Kjö03] and [Kjö04].

5.1 Additional Concepts

We have already mentioned Job-Trees (Section 3.4), as well as Application Modules AMs (Section 3.3), and in the following subsections we will take a closer look at the Job-Tree concept (Section 5.1.1) as well as a *Job-Tree Source* (JTS) (Section 5.1.2). We will also discuss how AMs are combined into a running system (Section 5.1.3).

5.1.1 Job-Trees

In Section 3.5, we said that a job begins with an `ENTER` statement for a buffered signal, and ends with an `EXIT` statement, and we also (in Section 3.4) informally defined "the set of jobs that originates from the

same external signal" as a Job-Tree. The 'external signal' is a (buffered) signal sent from a *Job-Tree Source* (which is explained in Section 5.1.2), and the "root" of the job-tree is the job J_1 started by this signal. The job-tree derived from J_1 is the least tree such that[3]

- J_1 is a node in the tree, and
- For every node (job) J_i in the tree that spawns off a new job J_k (which is done by the sending of a buffered signal, Section 3.4); J_k is also a node in the tree and there is an edge from J_i to J_k.

This means that a job-tree is the set of jobs, where the first job starts the others (by sending buffered signals). The job-tree terminates when all descendant jobs have terminated, i.e., when all jobs in the job-tree have executed 'their' `EXIT` statement without any preceding buffered signal sending statements. To 'visualize' the concept of job-trees, we complement Fig. 4 (showing the execution model of PLEX) with the corresponding job-tree as can be seen in Fig. 9.

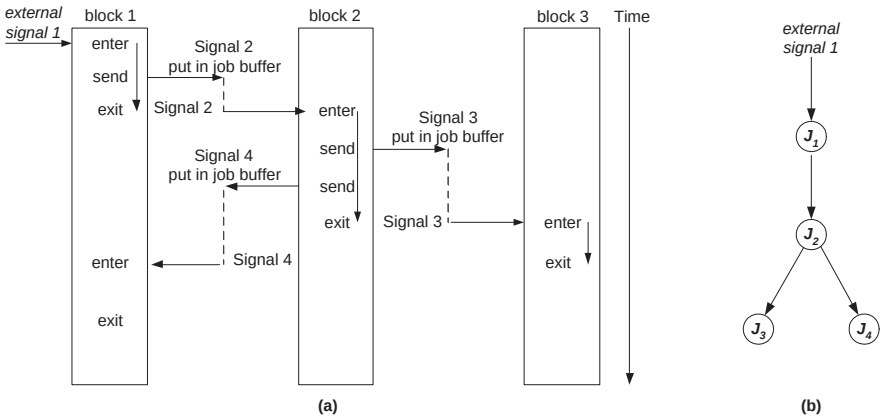

Figure 9: *(a) The "pseudo-parallel" execution model of PLEX (repeated from Fig. 4, together with the corresponding job-tree (b).*

Earlier in this report (Section 1) we said that the approach taken to achieve functional correctness was a restricted execution model which

[3]Note that we in this section give a general definition of a job-tree in terms of a graph. In Section 6 we will however use a list to represent a job-tree, since we need to preserve the order between individual jobs.

prevents some parts of the software to be executed in parallel. Those parts are jobs from the same job-tree, which are executed in the same sequential order as in the single-processor architecture. This restriction, and the way it is modeled, is discussed in Section 6.

5.1.2 Job-Tree Source

As we saw in the preceding section, a Job-Tree Source (JTS) is of importance when discussing job-trees since the signals from a JTS starts a new job-tree. JTS are mainly regional processors (RPs), Time Queues and Job Tables. Time Queues are special buffers for signals that are released (i.e., sent) at a specified time, whereas the Job Table is for jobs executed at short periodic intervals, e.g., incrementing clocks for time supervision. (For further reading, see [EL02].)

5.1.3 An AM Based System

In Section 3.3, we introduced the AM concept, and before we start describing the different execution paradigms in the following sections, we will say something about an AM based system (and we also recall from Section 3.3 that an AM consists of one, or several, PLEX blocks).

An AM based system, consists of the AMs (which forms the applications) together with some common resources. The common resources are collected into something called the *Resource Module Platform*, or RMP for short. As can be seen in Fig. 10, communication between different AMs is performed via something called AM protocol, whereas communication between an AM and the RMP is performed via ordinary signals (as described in Section 3.4).

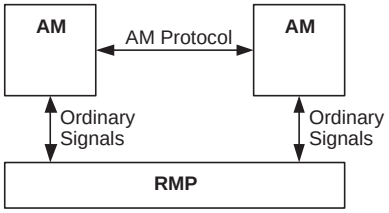

Figure 10: *Illustration of An AM based system (from [Kjö04]).*

5.2 FD: Functional Distribution

Functional Distribution, or FD for short, is an execution paradigm where the load sharing among the threads are achieved by pre-allocating each block to one of the threads, i.e., to *distribute the functions*. Each block does only exist in one instance, and once a block is allocated to a specific thread, it will always be executed by that thread. The term ***FD-mode*** refer to execution according to the FD principles (which is illustrated in Fig. 11).

In general, software that is to be executed in FD-mode may have to be treated in certain ways to preserve functional correctness since there may be situations when a specific (sequential) order, among parts of a program, is assumed.

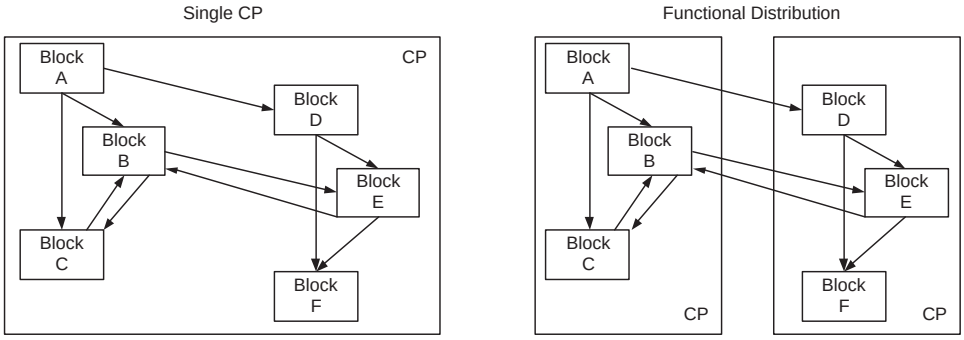

Figure 11: *Example (from [Kjö04]) on the FD principles: blocks, that in the single-pro. case is executed on the same CP, is in FD distributed over the available resources.*

5.3 CMX: Concurrent Multi-eXecutor

In contrast to the FD-mode execution, where each block is pre-allocated to one of the threads, no block is pre-allocated in CMX-mode. Instead, each block can be executed by any of the threads (as illustrated in Fig. 12). This means that since any of the threads can execute any block, it may very well be the case that two threads access the same block concurrently. To prevent data interference in such situations, lock handling is used, i.e., if a thread wants to execute a specific block, it must first acquire the corresponding lock, which on the other hand, may cause

dead-locks if nothing is done to prevent it.

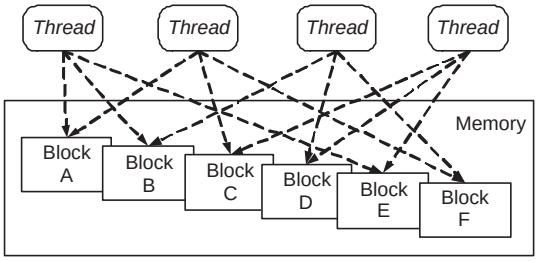

Figure 12: *The CMX paradigm, where any of the threads can execute any of the blocks that resides in memory.*

5.4 CMX-FD

The current execution model for parallel execution of PLEX, and the one modeled in Section 6, is CMX-FD. As hinted by the name, CMX-FD is the combination of the previous described execution paradigms FD and CMX. A prerequisite for the approach is an AM based system, but before we discuss the main ideas of the CMX-FD approach, we will make some additions to the AM concept (and the AM based system) that we discussed in Section 3.3 (and 5.1.3).

Now that we have discussed both Functional Distribution (FD), Section 5.2, and CMX, Section 5.3, we can add to the AM concept that an AM mainly consists of FD-blocks, together with a minor number of CMX-blocks, where the first type is allocated according to the FD principles, while the second type can be executed by any thread (i.e., according to the CMX principles). The same is true for the Resource Module Platform (RMP), i.e., that it consists of both FD- and CMX-blocks. The reason behind the different types of blocks is that some blocks (the CMX blocks) are reachable from different threads via a direct-signal[4] interface, which means that a signal to these blocks continues an ongoing job, and since a job is not allowed to leave the thread that executes it (Section 6), it must be possible for any thread to execute these blocks, which implies the shared memory. It should be stated that the CMX-mode

[4]These direct signals are in almost every case combined signals. (The different kind of signals was discussed in Section 3.4.)

would not be necessary if the blocks weren't reachable from different threads via direct signals, i.e., if all signals between different blocks were buffered.

The main idea behind the CMX-FD approach, illustrated in Fig. 13, is simply based on execution of as many blocks as possible in FD-mode, whereas the remaining blocks are executed in CMX-mode. Like in the FD-approach (Section 5.2), pre-allocation is used, but in CMX-FD it is the AMs, or more correct the FD-parts of the AMs, that are pre-allocated: each AM (i.e., the FD-part) is allocated to a thread according to a scheme given as initial configuration data (and, two or more AMs can be allocated to the same thread). The FD-mode blocks will always be executed by this thread, while we recall that CMX-mode blocks can be executed by any thread. However, with *Home thread* for a specific CMX-block, we denote the thread that its corresponding AM has been allocated to. This information will be of importance in Section 6.4 when we specify the semantics for buffered signals.

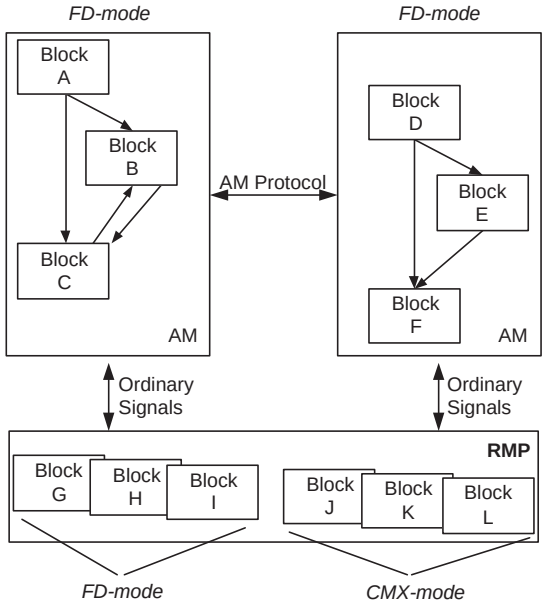

Figure 13: *Illustration of the CMX-FD execution paradigm.*

6 The Semantics

We begin this section with the abstract syntax for the modeled statements, together with some necessary definitions (Section 6.1), before specifying the state of the system (Section 6.2), and the parallel semantics (in the remaining sections).

6.1 Abstract Syntax

The semantics for PLEX will be given in terms of a semantics for the language *Core PLEX*, which is a simplified version of PLEX intended to capture its essential properties, namely the asynchronous communication, and the handling of jobs. Its basis is a simple imperative language with assignments, conditionals, and unstructured `GOTO`'s. The language also has a `SEND` statement to send direct or buffered signals, and an `EXIT` statement to terminate the current job.

Although simplified, it is actually possible (as we will show) to express many of the omitted PLEX statements in terms of already specified Core PLEX statements. We may therefore view the modeled language as the "Core" of PLEX.

Notable omissions from the real PLEX language, not modeled in terms of other Core PLEX statements, are the statements for signal reception (see below).

For modeling reasons, we have also introduced a statement not present in real PLEX; the `SKIP`-statement with its standard semantics

$$s \xrightarrow{\texttt{SKIP}} s$$

i.e., the execution of `SKIP` from an initial state s results in the same state s.

The abstract syntax for the modeled language is given in Table 1. Following [NNH05], we are using labeled statements, since we need labels to model program points to where control can be transferred. We assume that each label occurs only once which means that the programs are uniquely labeled, and since this is the case, we can, for a given Core PLEX program S, define the function $Stmt\colon \mathbf{Lab} \to \mathbf{Stmt} \cup \mathbf{BExp}$ by $Stmt(l) = S'$ (or b) precisely when S contains the statement $[S']^l$ (or condition $[b]^l$). Since the programs are uniquely labeled, we can also define the inverse to the function S, like $Stmt^{-1}\colon \mathbf{Stmt} \cup \mathbf{BExp} \to \mathbf{Lab}$

n	$\in$	**Num**, numerals
x	$\in$	**Var**, program variables
l	$\in$	**Lab**, labels
a	$\in$	**AExp**, arithmetic expressions
b	$\in$	**BExp**, boolean expressions
S	$\in$	**Stmt**, statements
op_a	$\in$	arithmetic operators
op_r	$\in$	relational operators
a	::=	$x \mid n \mid a_1\ op_a\ a_2$
b	::=	$a_1\ op_r\ a_2$
$data$	::=	$\{\mathbf{Var} \mid \mathbf{Num}\}^k \perp^{25-k}, \quad 1 \leq k \leq 25$
S	::=	$[x := a]^l \mid S_1;\ S_2 \mid [\texttt{GOTO}\ label]^l \mid \texttt{IF}\ [b]^l\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2 \mid$
		$[\texttt{SEND}\ signal]^l \mid [\texttt{SEND}\ signal\ \texttt{WITH}\ data]^l \mid [\texttt{EXIT}]^l \mid$
		$[\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label]^l_{label} \mid$
		$[\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label]^l_{label} \mid$
		$[\texttt{RETURN}\ cbsignal]^l \mid [\texttt{RETURN}\ cbsignal\ \texttt{WITH}\ data]^l \mid$
		$[\texttt{TRANSFER}\ signal]^l \mid [\texttt{TRANSFER}\ signal\ \texttt{WITH}\ data]^l$

Table 1: The abstract syntax for Core PLEX.

In Section 3.2, we said that the only way to access the code in a block is through its sub-programs, and since the entry points to the sub-programs are the signal receiving statements, we will simply regard a signal as an entry label to a block (and omit the statements for signal reception). Therefore, we define

$$\mathbf{ELab} \subseteq \mathbf{Lab}$$

as the set of *signal labels*. We need to distinguish between direct signals, and buffered signals, and we must also distinguish whether the latter are internal or external. To that end, we partition **ELab** into three disjoint sets **Dir**, **Buf**, **Ext** for the respective labels. Furthermore, we partition **Buf** into the disjoint sets **LevA**, and **LevB**, in order to capture the different priorities among the signals. (Recall from Section 3.4 that every signal is assigned a priority level.)

When defining the state transitions for the semantics, it then helps to have a flow graph-oriented description which defines successor labels. Therefore, we define three functions $succ$, $succT$, $succF$ from labels to labels. They are defined in the style of [NNH05], through the three functions $init$: **Stmt** $\rightarrow$ **Lab**, $final$: **Stmt** $\rightarrow P($**Lab**$)$, and $Flow$: **Stmt** $\rightarrow P($**Lab** $\times$ **Lab**$)$ in Table 2. Additionally, we also need to define the notion of *Interflow*, *IF*, in order to define $Flow(S)$ for the combined signal sending statement.

Definition 1 *For any Core PLEX program S, the partial functions $succ$, $succT$, $succF$: **Lab** $\rightarrow$ **Lab** are defined by:*

- $succ(l) = l'$ *if* $(l, l') \in Flow(S)$ *and* $(l, l'') \in Flow(S) \implies l'' = l'$, *otherwise undefined,*
- $succT(l) = init(S_1)$ *if* `IF` $[b]^l$ `THEN` S_1 `ELSE` S_2 *is a statement in* S, *otherwise undefined,*
- $succF(l) = init(S_2)$, *ditto,*

Definition 2 *For any Core PLEX program $S, Interflow$, is defined by:*

- $IF = \{\ (l, cfsig, l', label) \mid S$ *contains* $[$`RETURN` $cbsig]^{l'}$ *as well as* $[$`SEND` $cfsig$ `WAIT FOR` $cbsig$ `IN` $label]^l_{label}\}$

S	$init(S)$	$final(S)$	$Flow(S)$
$[\texttt{SKIP}]^l$	l	$\{l\}$	$\emptyset$
$[x := a]^l$	l	$\{l\}$	$\emptyset$
$S_1;\ S_2$	$init(S_1)$	$final(S_2)$	$flow(S_1) \cup flow(S_2) \cup$ $\{(l, init(S_2))\mid l \in final(S_1)\}$
$[\texttt{GOTO}\ label]^l$	l	$\emptyset$	$(l,\ label)$
$\texttt{IF}\ [b]^l\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2$	l	$final(S_1) \cup final(S_2)$	$flow(S_1) \cup flow(S_2) \cup$ $\{(l, init(S_1)), (l, init(S_2))\}$
$[\texttt{SEND}\ signal]^l$ **(**$signal \in$ **Dir)**	l	$\emptyset$	$(l,\ signal)$
$[\texttt{SEND}\ signal]^l$ **(**$signal \in$ **Buf)**	l	$\{l\}$	$\emptyset$
$[\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}$ $cbsig\ \texttt{IN}\ label]^l_{label}$	l	$\emptyset$	$(l, cfsig) \cup (l', label)$ $l' = Lab(\texttt{RETURN}\ cbsig)$
$[\texttt{RETURN}\ cbsignal]^l$	l	$\emptyset$	$\{(l,\ label)\mid (l', l'', l, label) \in IF\}$
$[\texttt{TRANSFER}\ signal]^l$	l	$\emptyset$	$(l,\ signal)$
$[\texttt{EXIT}]^l$	l	$\emptyset$	$\emptyset$

Table 2: Definition of $init$, $final$, and $Flow$. Note that since it is irrelevant for the definitions of the above functions whether or not a signal carry any data, we have omitted those cases from the above table.

Further on, we recall that the code (and the data) is structured in blocks (Section 3.2), and we assume that the program under consideration consists of β blocks. We then take each integer $1, \ldots, \beta$ to be the identifier for a unique block, and we define two functions

$$BV\colon \mathbf{Var} \to \{1, \ldots, \beta\}$$
$$BL\colon \mathbf{Lab} \to \{1, \ldots, \beta\}$$

which decide, for each program variable and program part, respectively, which block it belongs to. BV and BL induce partitionings of **Var** and **Lab**, respectively. Furthermore, we impose the following constraints to ensure that data accesses do not take place across block borders, and that program control is not transferred to some other block except through sending a signal. For all labels l in a Core PLEX program,

$$Stmt(l) \neq \texttt{SEND}\ signal \implies BL(succ(l)) = BL(l), \quad \text{if } succ(l) \text{ defined}$$
$$Stmt(l) \in \mathbf{BExp} \implies BL(succT(l)) = BL(succF(l)) = BL(l)$$
$$\forall x \in FV(Stmt(l)).BV(x) = BL(l)$$

Here, $FV(S)$ is the set of (free) variables in statement S.

Finally, we recall from Section 5.4 that each block is pre-allocated to one of the threads. For a system with β blocks, and k threads, we define the function

$$Alloc\colon \{1, \ldots, \beta\} \to \{1, \ldots, k\}$$

which for a given block determines which thread it has been allocated to. (We will use this information in Section 6.4 when discussing the semantics for the signal statements.)

6.2 The State of the System

Since the execution of statements are modeled as state transitions, and we recall (from Section 1) that the state we model is determined by the current parallel implementation of the real PLEX, as well as by the underlying architecture/execution model, we devote this section to the state of the system.

Whereas the sequential semantics only needed to consider 'one' state[5];

$$\langle \mathcal{VSC}, \sigma, JBA, JBB \rangle$$

[5] In [Eri03], we defined the state as the tuple $\langle \mathcal{VSC}, \sigma, JBA, JBB, JBC, JBD, JBR \rangle$, but since we will only consider execution on the traffic handling level (B) in this report, we omit those parts that don't effect this level.

the parallel semantics will need to consider 'several' states simultaneously; for a system with k threads, each parallel state is a k+1-tuple $\langle s_1, \ldots, s_k, s_G \rangle$, where each s_i (i = 1, ..., k) is a *local state* and s_G is a *global* (or *shared*) state. The states we consider will have the following appearance:

$$
\begin{aligned}
s_1 &= \langle \mathcal{VSC}_1, JBA, JBB_1, Locks_1, \mathcal{F}_1, \delta_1 \rangle \\
&\quad \in \mathbf{Lab} \times [(\mathbf{ELab}, data)] \times [(\mathbf{ELab}, data)] \times P(\mathbf{LVar}) \times [[\mathbf{ELab}]] \times [\mathbf{Lab}] \\
s_i &= \langle \mathcal{VSC}_i, JBB_i, Locks_i, \mathcal{F}_i,\ \delta_i \rangle \\
&\quad \in \mathbf{Lab} \times [(\mathbf{ELab}, data)] \times P(\mathbf{LVar}) \times [[\mathbf{ELab}]] \times [\mathbf{Lab}], \quad i = \{2, \ldots, k\} \\
s_G &= \langle \sigma, \sigma_L \rangle \in (\mathbf{Var} \to N) \times (\mathbf{LVar} \to \{0, 1\})
\end{aligned}
$$

This models a system where each local state s_i can be modified only by thread i, but where the global state can be modified by any of the threads. The reason for the explicit specification of local state s_1 is that the corresponding thread (T_1) is the only thread that are allowed to execute jobs of priority `A` (urgent operating system jobs) as well as of priority `C/D` (administrative jobs). Any other thread are only allowed to execute jobs of priority `B` (traffic handling). Also note that since the local part of the state s is associated with the threads, instead of the processors, we can leave the actual number of processors unspecified, and neither do we have to consider how many threads each processor executes. The remaining of this section will be devoted to examine each of the components in the above state.

- We recall (from the previous section) the unstructured nature of the language (the use of `GOTO`'s). For this reason, we have made the program counter explicit in the state; $\mathcal{VSC}$ is a *virtual statement counter* which points to the current statement to execute, i.e., $\mathcal{VSC}_i$ holds the local program counter for thread i.

  When $\mathcal{VSC}$ receives the value $\perp$, we denote a state which does not map to any statement. The corresponding thread goes idle, and waits for a new job to execute.

- JBx, where x = $\{A,\ B\}$ are sequences of entry (signal) labels to model the job buffers. We denote the set of finite sequences, with elements from some set X, by $[X]$, the empty sequence by ε, $x : s$ denotes the sequence with head x and tail s, and $s : x$ denotes the

sequence with the first elements from s and last element x.
The possible transmission of signal data is captured in the job buffers. We recall, from Table 1, that the signal data is 1 to 25 variables (or constant values) possibly followed by a number of $\perp$ (undefined values). The number 25 is equal to the number of physical registers available.

- the variables in the system are divided into two categories

$$\mathbf{Var} \;=\; RM \;\cup\; DS \;\text{ such that }\; RM \;\cap\; DS \;=\; \emptyset$$

to reflect that some variables (RM) are only used for temporary storage of data that are local to a job, whereas the other class of variables (DS) is the shared data that can be accessed by any job that enters the block. The scope rules for the data implies that the DS can be further divided into the following disjunct sets

$$DS \;=\; DS_1,\; \ldots,\; DS_\beta \;\text{ such that }\; DS_i \;\cap\; DS_j \;=\; \emptyset \text{ for any } i \neq j$$

The contents of the memory, is described by the state σ, and a single variable x by $\sigma(x)$. To restrict σ to only the temporary variables (for instance) we will use the notation $\sigma|_{RM}$. In some cases a temporary variable will be treated as containing an "empty" value, i.e., its value is unknown and can't be used. We will denote this "absence" of a value with $\perp$.

The notation $\sigma|_{RM_i} \mapsto data$ will later in this report be used to denote transfer of the signal data into the temporary storage, and it is used as an abbreviation for

$$\{x_\alpha \mapsto data_\alpha \mid x_\alpha \in RM_i \;\wedge\; 1 \leq \alpha \leq 25\}$$

- In Section 4, we said that the parallel 'prototype' uses a locking scheme to protect a block from being concurrently accessed by two different jobs (from different job-trees). Therefore, we introduce the set **LVar**, which is a set of β binary *lock variables* $L_1, \ldots, L_\beta$, distinct from any variables in **Var**. In the 'prototype', every block is guarded by one specific lock, but since one lock may guard several blocks, L_i may equal L_j for some i and j. When a job is about to execute code in a specific block, it will acquire the associated

lock, and during its execution, a job will collect one or several locks. Thus, in the local state s_i, $Locks_i$ is the set of locks currently acquired by i. Only the thread that holds L_γ can access block γ. For the global state s_G, σ_L holds the current state of the lock variables: $\sigma_L(L_\gamma) = 1$ exactly when $L_\gamma \in Locks_i$ for some i.

- Earlier (Section 5.1.1) we said that jobs from the same job-tree are executed in the same sequential order as in the single-processor case, which implies that we need to keep track of the different job-trees. A complicating factor is that at the termination of a job, the corresponding job-tree might migrate to another thread. To model this, the job-trees are made explicit in the program state

  - $\mathcal{F}$ is a list of job-trees, where each job-tree is a list of jobs. For each job-tree $[sig : T]$ holds that sig always is executed before any other job in T, as can be seen in Section 6.6. (The creation of a job-tree is captured in Section 6.6 as well.) The job-trees in $\mathcal{F}_i$ might have been generated at other threads, but will continue their execution on thread T_i.

    The basic elements (signals) are always the same in JBB_i and $\mathcal{F}_i$, but where each JBB is a list of signals, is the corresponding $\mathcal{F}$ a list of lists of signals. The purpose is to collect each job-tree in JBB_i in a separate list in $\mathcal{F}_i$.
  - The first element of $\mathcal{F}_i$ will always be the job-tree currently executing on thread T_i.
  - To denote the removal of job-tree JT from $\mathcal{F}$, we will write $\mathcal{F}$ - JT, and define the operator - on lists in the following way

$$
\begin{aligned}
&[\,] - l \;=\; [\,] \\
&l - [\,] \;=\; l \\
&a : l - a : l' \;=\; l - l' \\
&a : l - a' : l' \;=\; a(l - a' : l') \\
&a \neq a'
\end{aligned}
$$

- Finally, when specifying the semantics for a combined signal, we must ensure that we are able to maintain the proper nesting of send, and return points (see Section 3.4, and Fig. 7). We therefore add the context information δ to each local state. The idea is

simply to maintain a list of 'return-labels' where we "push" the current label when sending the combined forward signal, and "pop" it when sending the combined backward signal.

In the following sections, the semantics for Core PLEX is given in terms of transitions rules from state to state. For each rule we omit those parts of the state that are not modified by the transition, which typically means that only the local part s_i, for some thread i, and the global part s_G are visible in the transition rules. The transitions have the form

$$s \xrightarrow{S,\, i} s'$$

for a transition that affects the local memory of thread i, and where $Stmt(\mathcal{VSC}_i) = S$ (except for the rules, modeling the arrival of an external signal, as well as the rule for starting a new job, whose transitions are labeled with ϵ, see Section 6.6). When specifying the semantics, we will only consider the general case; execution on the Traffic handling level (priority B). The reason is that these are the jobs that are executed in parallel.

In an initial start up phase, the state would have the following contents:

$$\begin{aligned} s_1 &= \langle \bot, \varepsilon, \varepsilon, \emptyset, \varepsilon, \varepsilon \rangle \\ s_i &= \langle \bot, \varepsilon, \emptyset, \varepsilon,\ \varepsilon \rangle, \quad i = \{2, \ldots, k\} \\ s_G &= \langle \sigma|_{RM} \mapsto \emptyset|_{DS} \mapsto \Upsilon, \{L_\gamma \mapsto 0 \mid L_\gamma \in \sigma_L\} \rangle \end{aligned}$$

The initial state expresses that the local parts of the state are empty, i.e., the $\mathcal{VSC}_i$ does not map to any statement; the temporary storage (RM_i) is empty; the are no signals in the JBB_i job-buffer (which we recall is modeled as a list of signals); and there are no locks collected (as indicated by $\emptyset$ at the place for $Locks_i$). Job buffer A is empty as well, and each lock L_γ is available (i.e., no block is locked). The values of the variables in the Data Store (DS) are provided by the programmer, or loaded from external storage depending on if the system is re-started or not, and also on the different types of the variables. We will not discuss this further (instead we refer to [EL02] where this is discussed in more detail) more than to say that the variables in the DS always have some initial values Υ. $\mathcal{F}_i$ contains an empty value since no job has been started yet, and consequently there are no job-trees built either. This goes for δ_i as well, i.e., since no jobs has been executed, there are no

context information available.

6.3 The Semantics for the Basic Statements

Starting in this section, we will specify the semantics for parallel execution of Core PLEX in the architecture/execution model that was described in Section 4 and 5.4. We begin with what we call the *basic statements*, i.e., assignments[6], jump-statements, conditionals, and iterations, and we continue with the semantics for the signal statements in Section 6.4.

If nothing else is stated, the transitions will be given for thread T_i where $i = \{2, \ldots, k\}$. The corresponding transitions exist for T_1 as well. However, we have chosen not to show them since they are almost identical (except for JBA in the local state s_1).

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{x:=a,\ i}$$

$$\langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma],\ \sigma_L \rangle \rangle$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{x:=a,\ i}$$

$$\langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma[x \mapsto \mathcal{ST}[\![st]\!]\sigma],\ \sigma_L \rangle \rangle$$

We continue with the "ordinary" `IF-THEN-ELSE` construct

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i}$$

$$\langle succT(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{tt}$$

[6] Obviously, in any kind of assignment, the types of the variables need to match each other. We will assume that this is the case (and rely on that the compiler detects any kind of violation to this).

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i}$$

$$\langle succF(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{ff}$$

We also note that there is a "shortened" version of the IF-THEN-ELSE construct; IF b THEN S_1. However, this statement can be expressed in terms of the above specified IF-THEN-ELSE statement if we take

$$S_2 = \texttt{SKIP}$$

The IF statement are followed by the GOTO statement, which could be both conditional and unconditional. The semantics for the unconditional GOTO is specified as

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{GOTO}\ label,\ i}$$

$$\langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

For the conditional GOTO statement, IF b GOTO $label$, we note that with

$$S_1 = \texttt{GOTO}\ label, \quad \textbf{and} \quad S_2 = \texttt{SKIP}$$

this statement, similarly to the above "shortened" IF-THEN-ELSE construct, can be expressed in terms of the already specified
IF b THEN S_1 ELSE S_2 statement!

Next, we have the statement for selection (CASE) which in many ways are similar to the SWITCH statement in C. The CASE statement has the general form

$$\texttt{CASE}\ expression\ \texttt{IS}\ \{\texttt{WHEN}\ choice\ \texttt{DO}\ S\}^+\ \texttt{OTHERWISE}\ \texttt{DO}\ S_n$$

where the $\{\texttt{WHEN}\ choice\ \texttt{DO}\ S\}^+$ part can be repeated any number of times. When used by the programmer, the statement is written in the following manner;

$$\begin{array}{ll} \texttt{CASE}\ expression\ \texttt{IS} & \texttt{WHEN}\ choice_1\ \texttt{DO}\ S_1 \\ & \texttt{WHEN}\ choice_2\ \texttt{DO}\ S_2 \\ & \ldots \\ & \texttt{OTHERWISE DO}\ S_n \end{array}$$

and similarly to the above specified conditional GOTO-, and the shortened IF-statements, we can express the CASE statement in terms of the IF-THEN-ELSE statement in the following way;

$$\texttt{IF}\ [expression = choice_1]^l\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2^{'},\ \textbf{where}$$

$$S_2^{'} = \texttt{IF}\ [expression = choice_2]^{l'}\ \texttt{THEN}\ S_2\ \texttt{ELSE}\ S_3^{'},\ \textbf{and}$$

$$S_{n\text{-}1}^{'} = \texttt{IF}\ [expression = choice_{n\text{-}1}]^{l''}\ \texttt{THEN}\ S_{n\text{-}1}\ \texttt{ELSE}\ S_n$$

The last rules in this section is concerned with the different iteration statements that are available in PLEX. From [Eri03], we know that the well known While statement is missing in PLEX. The main reason is that this construct may give rise to unpredictable execution times, something that should be avoided in a Real-Time system[7]. Instead, PLEX offers three different statements for iteration which are all used for scanning files or indexed variables between given start and stop values.

The general form of the first statement, ON, is one of the following

$$\texttt{ON}\ pointer/variable\ \texttt{FROM}\ expression_1\ \texttt{UPTO}\ expression_2\ \texttt{DO}\ S$$

$$\texttt{ON}\ pointer/variable\ \texttt{FROM}\ expression_1\ \texttt{DOWNTO}\ expression_2\ \texttt{DO}\ S$$

where the statement S is executed a number of times (i.e., until $expression_1$ equals $expression_2$). And similar to some of the discussed statements above, we can express these statements in terms of already specified statements. With the assumption that i is a variable not already used by some code, we can re-write the first statement in the following way

[7] The AXE system has been classified as a soft Real-Time system by Arnström et. al in [AGG99].

i = $expression_1$
$LFalse$) IF i = $expression_2$ THEN GOTO $LTrue$
S
i = i+1
GOTO $LFalse$
$LTrue$) *remaining statements*

The re-writing for the second case is analog, simply replace i = i+1 with i = i-1 in the above code. This re-writing does in fact mimic the behavior of a standard compiler generating intermediate code for a corresponding WHILE loop[8].

The second iteration statement, FOR ALL, which iterates from $expression_1$ down to $expression_2$ (which can be omitted if it is 0)

FOR ALL *pointer/variable* FROM $expression_1$ UNTIL $expression_2$ DO S

is expressed in the same way as the ON...DOWNTO... statement;

i = $expression_1$
$LFalse$) IF i = $expression_2$ THEN GOTO $LTrue$
S
i = i-1
GOTO $LFalse$
$LTrue$) *remaining statements*

The last statement for iteration, FOR FIRST, is similar to the FOR ALL statement, except that the loop is aborted as soon as the conditional part is fulfilled.

FOR FIRST *pointer/variable* FROM $expression_1$ UNTIL $expression_2$ WHERE

condition IS CHANGED TO $expression_3$ DO S

The FOR FIRST statement is expressed as:

i = $expression_1$
$LStart$) IF i = $expression_2$ THEN GOTO $LDone$
IF *variable* = $expression_3$ THEN GOTO $LNext$
i = i-1
GOTO $LStart$
$LNext$) S
$LDone$) *remaining statements*

[8] See for instance [ASU86]

6.4 The Semantics for the Signal Statements

Before we continue with the semantics for the different signal statements, we recall (from Section 6.1) that we regard a signal as an entry label to a block, and that we have defined $\mathbf{ELab} \subseteq \mathbf{Lab}$ as the set of *signal labels*. Further more, **ELab** has been partitioned into the disjoint sets **Dir**, **Buf**, **Ext** in order to distinguish between direct, buffered, and external signals.

Regarding the receiver of a specific signal (i.e., the receiving thread), the following applies[9]:

Direct signals: since a job is **not** allowed to leave the thread that starts executing it (which further motivates our decision to associate the local parts of the state s with the threads, and not with the processors, as we did in Section 6.2), the signal sending statement, and the code that is executed as a result of the signal sending, will **always** be executed by the same thread.

Buffered signals: for buffered signals the situation is slightly different since we have to consider if the receiving block is an FD-mode or a CMX-mode block. Since FD-mode blocks always are executed by the thread that they were allocated to (see Section 5.4), a buffered signal to an FD-mode block will be received by this thread, i.e., the signal is placed in the job buffer associated with the thread in question.

To answer the question of which thread that receives a buffered signal sent to a specific CMX block, we have to point out that there are three different types of CMX-mode blocks, where the type of the block determines where the signal is to be buffered. Which buffer that is to receive the buffered signal (which also means that the corresponding thread will execute the block) is given by Table 3, where we also see when information about the *Home thread* for a given CMX-mode block (which we discussed in Section 5.4) is of importance.

[9] We recall (Section 3.4) that from a semantical point of view, the main distinction is between direct and buffered signals; a direct signal continues an ongoing job whereas a buffered one spawns off a new job.

Buffered signal sent to:	**Buffered signal received by:**
CMX-mode block, Type-1	*buffer associated with the sending thread*
CMX-mode block, Type-2	*buffer associated with the Home thread*
CMX-mode block, Type-3	*buffer associated with thread* as specified by initial configuration data.

Table 3: *Receiving buffers for buffered signals to CMX-mode blocks.*

As we have seen, a buffered signal will be executed either by the thread its corresponding block has been allocated to (in case of an FD-mode block, or a CMX-mode block of Type 2), the thread that sends the signal (in case of CMX Type 1), or by the thread specified in the configuration data (CMX Type 3). This means that as soon as we know the execution mode of the receiving block we will also know which thread that will execute the buffered signals sent to that block. Therefore, we define the function

$$Type\colon \{1,\ldots,\beta\} \rightarrow \{FD, CMX_1, CMX_2, CMX_3\}$$

which for a given block determines the execution mode for the same block.

Now, since we can determine both the execution mode as well as the thread a given block has been allocated to, we can determine the receiver of any buffered signal (i.e., the thread that will execute the signal). To do this, we define the function

$$Receiver\colon \mathbf{ELab} \rightarrow \{1,\ldots,k\}$$

in the following way

$$Receiver(signal) = \begin{cases} Alloc(BL(signal)) & \textbf{if } Type(BL(signal))=FD \\ i & \textbf{if } Type(BL(signal))=CMX_1 \\ Alloc(BL(signal)) & \textbf{if } Type(BL(signal))=CMX_2 \\ \Upsilon & \textbf{if } Type(BL(signal))=CMX_3 \end{cases}$$

where i = id of the sending thread, and Υ = value specified in configuration data.

However, we must emphasize that *a buffered signal (of priority* `B`*) sent from thread* T_i *always is buffered in* T_i*s own job buffer, at a first step, before the signal is moved to a job buffer according to the above scheme.* The reason is the restricted execution model (which we discussed in Section 5.1.1) where jobs from the same job-tree are prevented from being executed concurrently. We will later in this section (in the rules for sending a buffered signal), and in Section 6.5, see how we deal with the described restrictions.

Finally, the handling of lock variables in the transitions models the lock handling in the parallel 'prototype'. The transitions for sending a direct signal attempt to transfer control to a possibly new block, but will not be enabled unless the corresponding lock is free. In the transition, the executing thread will then atomically take the lock. For the termination of a job (`EXIT`), the transitions are divided in two parts: one transition for `EXIT` which releases all the locks held by the thread, and one 'job'-transition that can succeed when the lock of the block is free. The effect of this is that locks are successively collected by a job, and then released all together when the job terminates.

With the above discussion, we are ready to approach the semantics for the signal sending statements, and we start with the semantics for the single[10] signals

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i}$$

$$\langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Dir},\ \gamma = LB(signal),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i}$$

$$\langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Dir},\ \gamma = LB(signal),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

[10] The single signals do not, in contrast to the combined signals, require a reply. For a discussion about the different signal properties, see Section 3.4.

Before we continue, we recall (from Section 3.2) that there are different priorities among the jobs, and in Section 6.5 we will discuss how we have to consider ongoing activities in the system when a new job is to be started (i.e., when a buffered signal is to be fetched from a buffer).

The following rules deal with the sending of a buffered signal. The first two cases deals with the sending of a priority A signal, which is immediately inserted in JBA of s_1. (Note that we in this case have to consider two local states; s_1, and s_i.) The last two cases are the general cases, i.e., a signal of priority B inserted at JBB_i of s_i (where $i = \{2, \ldots, k\}$).

We would also like to stress that the statement for sending a buffered signal is currently subject to change. The proposed change is discussed in Section 6.4.1.

$$\langle s_1, \ldots, s_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i} \langle s'_1, \ldots, s'_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

$$\begin{aligned} where \quad & s_1 = \langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1 \rangle \\ & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_1 = \langle \mathcal{VSC}_1,\ JBA : (signal, \perp),\ JBB_1,\ Locks_1,\ \mathcal{F}_1 : [signal],\ \delta_1 \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \end{aligned}$$

$$\text{if} \quad signal \in \mathbf{Buf}, \quad signal \in \mathbf{LevA}$$

$$\langle s_1, \ldots, s_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i} \langle s'_1, \ldots, s'_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

$$\begin{aligned} where \quad & s_1 = \langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1 \rangle \\ & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_1 = \langle \mathcal{VSC}_1,\ JBA : (signal, data),\ JBB_1,\ Locks_1,\ \mathcal{F}_1 : [signal],\ \delta_1 \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \end{aligned}$$

$$\text{if} \quad signal \in \mathbf{Buf}, \quad signal \in \mathbf{LevA}$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [\mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i}$$

$$\langle succ(\mathcal{VSC}_i), JBB_i : (signal, \perp),\ Locks_i,\ [\mathcal{T} : signal] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevB}$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i}$$

$$\langle succ(\mathcal{VSC}_i), JBB_i : (signal, data),\ Locks_i,\ [\mathcal{T} : signal] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevB}$$

The concept of combined signals is shown in Fig. 14, and we recall from Section 3.4 that the thing that distinguish a combined signal from other direct signals[11] is that the combined signal always requires an answer (a reply signal).

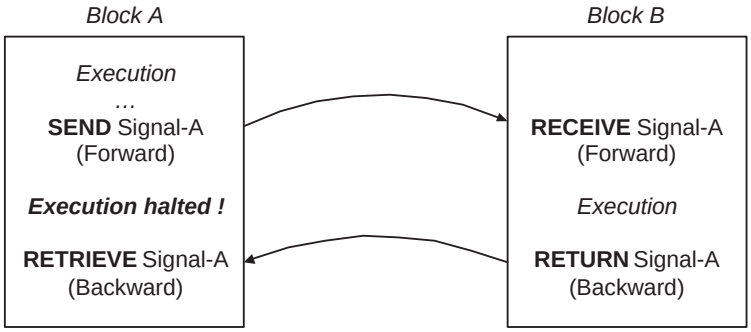

Figure 14: *The PLEX statements for sending/receiving combined signals. Note that the signal receiving statements is omitted in Core PLEX (see Section 6.2).*

[11] A combined signal, as well as a local signal, is always direct!

The semantics for the combined signals are as follows

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label,\ i}$$

$$\langle cfsig, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \quad \gamma = LB(cfsig),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label,\ i}$$

$$\langle cfsig, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \quad \gamma = LB(cfsig),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

$$\langle \mathcal{VSC}_i, JBB_i, Locks_i,\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{RETURN}\ cbsig,\ i}$$
$$\langle label, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L \rangle \rangle$$

$$\langle \mathcal{VSC}_i, JBB_i, Locks_i,\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{RETURN}\ cbsig\ \texttt{WITH}\ data,\ i}$$
$$\langle label, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L \rangle \rangle$$

We end this section with the semantics for the local signals, and as was said in Section 3.4, the difference between local and non-local signals is that the former is sent between entities in the same block, whereas the latter is sent between entities in different blocks. This means that no variable values are destroyed by a local signal statement, which is the case with non-local signals (where the variables in the Register Memory (RM) are destroyed).

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{TRANSFER}\ signal,\ i}$$
$$\langle signal, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{TRANSFER}\ signal\ \texttt{WITH}\ data,\ i}$$
$$\langle signal, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data \setminus \bot, \sigma_L \rangle \rangle$$

6.4.1 A New Buffered Signal

In conjunction with the semantics for the sending of a buffered signal, we mentioned (on page 34) that the statement is currently subject to change. In its current 'version' the sending of a buffered signal results in a new job within the same job-tree, whereas with the new, proposed/suggested, extension it should be possible to use a specific keyword (**JOBTREE** (?)) with a new job-tree as the result.

This will increase the level of parallelism since there would be no sequential order to maintain between the job that sends the signal, and the job that will be the result of the signal. For this reason, this new buffered signal can be sent directly to its destination (i.e., be put in the appropriate buffer) instead of being put in the buffer of the sending thread as the 'ordinary' buffered signal are done (to maintain the previously described sequential order).

The semantics for the new buffered signal is as follows:

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE},\ i}$$

$$\langle succ(\mathcal{VSC}_i), JBB_i : (signal, \bot),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Buf},\ Receiver(signal) = i$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE WITH}\ data,\ i}$$

$$\langle succ(\mathcal{VSC}_i), JBB_i : (signal, data),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

$$\text{if} \quad signal \in \mathbf{Buf},\ Receiver(signal) = i$$

In the following two rules, we must consider two local states (s_i and s_j) in the case $i \neq j$ since they model the case when the buffered signal is sent directly to its destination. As in the other cases, we show only those parts of the state $\langle s_1, \ldots, s_k, s_G \rangle$ that are effected of the transition (in this case s_i, and s_j).

$$\langle \ldots, s_i, , s_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE},\ i} \langle \ldots, s'_i, , s'_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

$$\begin{aligned} where \quad & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s_j = \langle \mathcal{VSC}_j,\ JBB_j,\ Locks_j,\ \mathcal{F}_j,\ \delta_j \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_j = \langle \mathcal{VSC}_j,\ JBB_j : (signal, \bot),\ Locks_j,\ \mathcal{F}_j : [signal],\ \delta_j \rangle \end{aligned}$$

$$\text{if} \quad signal \in \mathbf{Buf},\ Receiver(signal) = j \neq i$$

$$\langle \ldots, s_i, , s_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE WITH}\ data,\ i} \langle \ldots, s'_i, , s'_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

$$\begin{aligned} where \quad & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s_j = \langle \mathcal{VSC}_j,\ JBB_j,\ Locks_j,\ \mathcal{F}_j,\ \delta_j \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_j = \langle \mathcal{VSC}_j,\ JBB_j : (signal, data),\ Locks_j,\ \mathcal{F}_j : [signal],\ \delta_j \rangle \end{aligned}$$

$$\text{if} \quad signal \in \mathbf{Buf},\ Receiver(signal) = j \neq i$$

6.5 The Semantics for the `EXIT` Statement

As we saw in Section 3.5, the `EXIT` statement is of importance since it marks the termination of an ongoing job, but we have also indicated (in Section 6.4, on page 33) that the effect of executing an `EXIT` statement is the start of a new job. However, the semantics for the `EXIT` statement below is only concerned with termination of the ongoing job (i.e., releasing of the locks collected by the job). The transition for starting a new job is postponed to the following section. The reason for this division is the lock handling mechanism; the transition for the `EXIT` statement releases all locks held by the thread, whereas the transition for starting a new job can succeed when the lock of the block is free, and then lets the thread acquire the lock. Without this division, other threads would never be allowed to execute code in the block.

The below rules for the EXIT statement models (1) the termination of the currently executed job-tree, (2) that the job-tree hasn't terminated and will continue its execution on T_i, and (3) that the job-tree migrates to T_j. Note that the last rule (when the job-tree migrates) must consider two local states; s_i and s_j.

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [\,] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{EXIT},\ i}$$

$$\langle \bot, JBB_i, \emptyset,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle \rangle$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{EXIT},\ i}$$

$$\langle \bot, JBB_i, \emptyset,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle \rangle$$

$$\text{if} \quad Receiver(signal) = i$$

$$\langle \ldots, s_i, , s_j, \ldots, s_G \rangle \xrightarrow{\texttt{EXIT},\ i} \langle \ldots, s'_i, , s'_j, \ldots, s'_G \rangle$$

$$\begin{aligned}
where \quad & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i \rangle \\
& s_j = \langle \mathcal{VSC}_j,\ JBB_j,\ Locks_j,\ \mathcal{F}_j,\ \delta_j \rangle \\
& s_G = \langle \sigma, \sigma_L \rangle \\
& s'_i = \langle \bot, JBB_i - \{(signal, data) : \mathcal{T}\},\ \emptyset,\ \mathcal{F}_i,\ \delta_i \rangle \\
& s'_j = \langle \mathcal{VSC}_j, JBB_j : (signal, data) : \mathcal{T},\ Locks_j,\ \mathcal{F}_j : [signal : \mathcal{T}],\ \delta_j \rangle \\
& s'_G = \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle
\end{aligned}$$

$$\text{if} \quad Receiver(signal) = j \ \neq \ i$$

6.6 Additional transitions

The following rule deals with the start of a new job. The transition succeeds when the lock of the corresponding block is free, **and** if job buffer A of local state s_1 is empty. The second condition models the fact

that jobs on traffic level (priority B) must wait for jobs of priority A. (The different levels of priority among jobs were discussed in Section 6.2.)

$$\langle \bot,\ (signal, data) : JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$$

$$\langle signal, JBB_i, \{L_\gamma\},\ [\mathcal{T}] : \mathcal{F}_i - [signal : \mathcal{T}],\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

$$\text{if } \ \gamma = LB(signal),\ \sigma_L(L_\gamma) = 0,\ s_1(JBA) = \varepsilon$$

The last transitions models the insertion from the environment of an external signal into a job queue. Note that the external signal can be of priority A or priority B. In the first case, the external signal is inserted in JBA of s_1. The second case is the general case, i.e., priority B inserted in JBB_i of s_i (where $i = \{2, \ldots, k\}$). The rules are always enabled, and they introduce nondeterminism into the semantics:

$$\langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$$

$$\langle \mathcal{VSC}_1,\ JBA : (signal, data),\ JBB_1,\ Locks_1,\ \mathcal{F}_1 : [signal],\ \delta_1,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

$$\text{if } signal \in \mathbf{Ext}, \ \ signal \in \mathbf{LevA}$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$$

$$\langle \mathcal{VSC}_i,\ JBB_i : (signal, data),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

$$\text{if } signal \in \mathbf{Ext}, \ \ signal \in \mathbf{LevB}$$

6.7 Global Transitions

In the previous sections we have defined the meaning of each individual Core PLEX statement, when executed 'locally' by thread T_i, as in

$$s \xrightarrow{S,\ i} s'$$

However, the description is not complete as long as we don't consider the concurrently executing threads.

We recall (from Section 6.2) that the state we consider is defined by the tuple $\langle s_1, \ldots, s_k, s_G \rangle$, where each s_i (i = 1, ..., k) is a local state and s_G is a global state that can be modified by any of the threads.

The following rules, valid for any i where $0 \leq i \leq k$, specify the global transitions[12]

$$\frac{s_i \rightarrow s_i'}{\langle s_1, \ldots s_i, \ldots, s_k, s_G\rangle \rightarrow \langle s_1, \ldots s_i', \ldots, s_k, s_G\rangle}$$

$$\frac{s_i \rightarrow s_i'}{\langle s_1, \ldots s_i, \ldots, s_k, s_G\rangle \rightarrow \langle s_1, \ldots s_i', \ldots, s_k, s_G'\rangle}$$

The rules state that whenever there is a local transition at thread T_i, there is a corresponding global transition that only affects the local part of the state s_i (first case), or that affects the local, as well as the global, part of the state (second case).

7 Summary

In previous works, we have presented a small-steps operational semantics for the language PLEX. These works modeled the execution on the current single-processor architecture. In this report, we extend our previous works by specifying a restricted parallel semantics for the language Core PLEX, which is a simplified version of the real PLEX language. The semantics in this report models an attempt to execute existing PLEX code, without modifications, on a parallel architecture.

The architecture under consideration is a multi-threaded shared-memory architecture. The parallel architecture, and its execution model, is designed to be 'functionally equivalent' with the single-processor system. The approach taken to achieve this 'equivalence' is a restricted execution model, which (by a locking mechanism) prevents some parts of the programs to be executed concurrently.

A more aggressive parallelization would allow these activities to execute in parallel, but parallel execution most likely 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 functional equiva-

[12] The transitions are of standard form as used for instance in [Win93]

lence is preserved. To ensure the correctness of such criteria, the formal semantics of the language has to be considered.

Future work includes a case study of possible shared memory conflicts in the existing PLEX code, as well as deriving criteria for safe parallel execution.

8 Acknowledgements

This work has been supported by Ericsson AB, and Vinnova through the ASTEC competence centre. We want to thank Janet Wennersten and Ole Kjøller at Ericsson AB for technical support and discussions regarding PLEX and its implementations.

References

[AGG99] A. Arnström, C. Grosz, and A. Guillemot. GRETA: a tool concept for validation and verification of signal based systems (e.g. written in PLEX). Master's thesis, Mälardalen University, 1999.

[Ard97] M. A. Ardis. Formal Methods for Telecommunication System Requirements: A Survey of Standardized Languages. *Annals of Software Engineering*, 3:157–187, 1997.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ulman. *Compilers Principles, Techniques and Tools*. Addison-Welsey Publishing Company, 1986.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[CS01] M. Calder and C. Shankland. A Symbolic Semantics and Bisimulation for Full LOTOS. In *Proceedings of the 21st Inter-*

national Conferenence on Formal Techniquess for Networked and Distributed Systems, pages 185–200. IFIP, 2001.

[Däc00] Bjarne Däcker. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Licentiate thesis, Royal Institute of Technology, KTH, Sweden, 2000.

[EL02] J. Erikson and B. Lindell. The Execution Model of the APZ/PLEX - An Informal Description. Technical report, Mälardalen University, 2002.

[EL04] 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.

[Eri03] J. Erikson. A Structural Operational Semantics for PLEX. MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-166/2004-1-SE, Mälardalen University, 2003.

[Fre01] L. Fredlund. *A Framework for Reasoning About ERLANG Code*. PhD thesis, Royal Institute of Technology, KTH, Sweden, 2001.

[GGP03] 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.

[Hed98] Pekka Hedqvist. A parallel and multithreaded ERLANG implementation. Master's thesis, Computing Science Department, Uppsala University, Uppsala, June 1998.

[IT82] ITU-T. *CHILL: Formal Definition*, 1982. International Telecommunication Union, Volume 1, Part 1, 2, 3.

[IT99] ITU-T. *CHILL: The ITU-T Programming Language*, 11 1999. International Telecommunication Union, Geneva, (Recommendation Z.200).

[Kjö03] O. Kjöller. *CMX-FD - A TLP Execution Model with Functional Distribution*. Internal Technical Report, Ericsson AB, 2003.

[Kjö04] O. Kjöller. *CMX-FD Configuration*. Internal Technical Report, Ericsson AB, 2004.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2nd Edition*. Springer, 2005.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.

[SU05] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. In *Proc. 2nd Workshop on Structural Operational Semantics, SOS 2005*, July 2005.

[TG97] J. Thees and R. Gotzhein. A Formal Syntax and a Formal Semantics for Open Estelle. Technical Report 292/97, University of Kaiserslautern, 1997.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[Win00] Jürgen F. H. Winkler. CHILL 2000. *Telektronikk*, 96(4):70–77, 2000.