Mälardalen University Licentiate Thesis
No.85

Operational Semantics for PLEX

A Basis for Safe Parallelization

Johan Lindhult

May 2008

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-85485-80-2
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

The emerge of multi-core computers implies a major challenge for existing software. Due to simpler cores, the applications will face decreased performance if not executed in parallel. The problem is that much of the software is sequential.

Central parts of the AXE telephone exchange system from Ericsson is programmed in the language PLEX. The current software is executed on a single-processor architecture, and assumes non-preemptive execution.

This thesis presents two versions of an operational semantics for PLEX; one that models execution on the current, single-processor, architecture, and one that models parallel execution on an assumed shared-memory architecture. A formal semantics of the language is a necessity for ensuring correctness of program analysis, and program transformations.

We also report on a case study of the potential memory conflicts that may arise when the existing code is allowed to be executed in parallel. We show that simple static methods are sufficient to resolve many of the potential conflicts, thereby reducing the amount of manual work that probably still needs to be performed in order to adapt the code for parallel processing.

To Cina, Therése, and Simon

Acknowledgements

First of all, my deepest thanks goes to my supervisors Björn Lisper and Jan Gustafsson at Mälardalen University, as well as Janet Wennersten and Ole Kjöller at Ericsson.

This work has been supported by Ericsson AB, and Vinnova through the ASTEC competence center. Additional funding has been provided by ARTES, and SAVE-IT. Thank you all.

I would also like to take the opportunity to thank the following past and present colleagues; everybody at the Computer Science Lab at Mälardalen University, Markus Bohlin at SICS, everybody (including Patrik Thunström and Aminur Rahman Faisal) at FTE/DDM at Ericsson. Also Mats and Lars Winberg at Ericsson. An extra thanks to my former room-mate at Mälardalen university, Jan Carlson, with whom I have had a lot of discussions during my research (not to mention all the help I have got with LaTeX!).

No research is possible without a great administration. Thank you Harriet, Monika, and Else-Maj.

A special thanks to Peter Funk, Janet Wennersten (again), and Bosse Lindell.

A very warm thanks to the following friends; Waldemar Kocjan, Lars Bruce, ”DIF-Håkan” Persson, and Torbjörn Johansson.

Last, but certainly not least, I had never gone this far without the love and support from my wife Cina, and my children Therése and Simon. Also my parents (P-O and Kjerstin) as well as my brothers (Micke and Lasse) ”deserves” a thanks.

Johan Lindhult
Sala, April, 2008

Contents

Chapter 1

Introduction

Over the years, software in general has been benefiting from ever increasing clock speeds on new CPU's, but with the emerge of multi-core architectures this might have come to an end. When the individual cores becomes simpler, with lower clock speeds, in order to reduce power consumption, the software might (in the worst case) end up running slower on these new architectures. To fully utilize the capacity of such an architecture, different parts of the application need to be executed in parallel. The problem with much of todays software is that is sequential, i.e., the designer has assumed sequential execution.

Sequential software could (of course) be executed on a parallel architecture if the execution is sequential (as in Fig. 1.1 (a)), but that is a poor utilization of the possibilities of the architecture.

A general, and desirable, solution is automatic parallelization. Here, the programmer writes his/her program in a conventional, sequential language, and leaves all the "dirty work" to an optimizing compiler that transforms the sequential program into a parallel one. The "traditional" area of application for an optimizing compiler has been scientific applications, where an increasing processor capacity has an major impact on the performance since much of the work in these applications can be done in parallel. These applications are often written in languages like FORTRAN or C. Typical cases where parallelization has been applied is loops and accesses of arrays. A loop may have sub-parts, without any dependency among them, which could be executed in parallel. In the case of array access, it might be the case that different parts of a program (or different threads) access different parts of the array. The literature contains several surveys on automatic parallelization of sequential languages [1, 2, 3, 4].

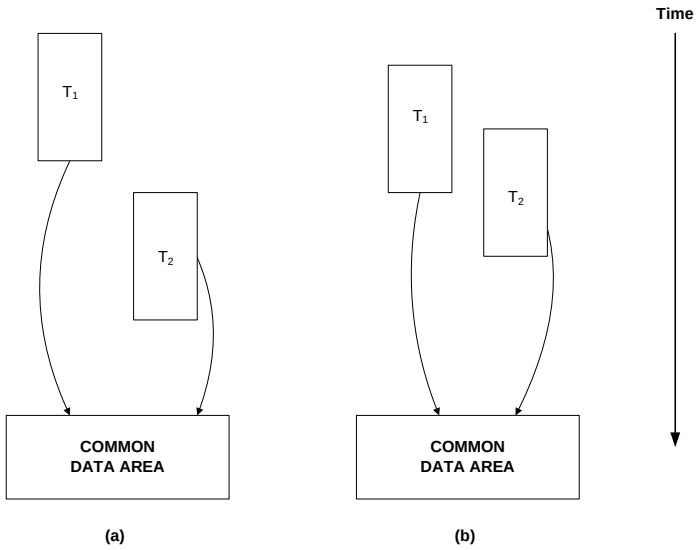

Figure 1.1: *Independent tasks with some common data; sequentially executed (a), or executed in parallel (b), where the different tasks* ***may*** *access the same data concurrently.*

Nevertheless, since new machines will increasingly be parallel [5, 6], software developers, and maintainers, still need to deal with concurrency one way or another in situations where the code can't be parallelized the "traditional" way.

For a large class of computer systems, the software has also been designed under the (implicit) assumption that activities in the system are executed on a non-preemptive basis. 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. Non-preemptive execution gives exclusive access to shared data, which guarantees that the consistency of such data is maintained.

However, on a parallel architecture, non-preemptive execution does not protect the shared data any longer since activities executed on different processors may access and update the same data concurrently, as in Fig. 1.1 (b). 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?*

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. PLEX jobs are executed in a priority-based, non-interruptible (at the same priority level), fashion on a single-processor architecture, and the language lacks constructs for synchronization. Due to the atomic nature of PLEX jobs (further discussed in Chapter 2.4), they can be seen as a kind of transactions. Thus, when executing them in parallel, one will face problems that are similar to maintaining the ACID[1] properties when multiple transactions, in a parallel database, are allowed to execute concurrently.

1.1 Research Questions

The primary motivation for our research is the fact that multi-core architectures will become a de-facto standard in a near future, while at the same time, there

[1] Atomicity = To the outside world, the transaction happens indivisibly, Consistency = The transaction does not violate system invariants, Isolation = Concurrent transactions do not interfere with each other, and Durability = Once a transaction commits, the changes are permanent [7].

are millions of lines of legacy event-based code in industry[2] . Rewriting this code into explicitly parallel code would be extremely expensive. Thus, there is a need to investigate methods to safely migrate such code to parallel architectures to get a maximum of efficiency gain and with a minimum of manual rewriting. By safe, we mean that the semantics of the PLEX jobs is preserved. Our general research question can then be formulated as:

Q: *Can different PLEX jobs execute in parallel, without changing the semantics of the system?*

which gives rise to the following, more detailed, questions

Q1: *How can we decide whether two PLEX jobs can be executed in parallel with preserved semantics?*

Q2: *Are there safe methods (e.g., program transformation) to increase the number of PLEX jobs that can be executed in parallel?*

1.2 Approach

To answer our first question, Q1, we believe that the specification of a program analysis that can classify parallel execution as safe (or unsafe) is a suitable way to go. Since we have defined safety as ”preserving the semantics”, the question is under what conditions the semantics is preserved? Since two PLEX jobs can only affect each other through shared data, a sufficient condition is if the shared data is kept consistent.

A case study of the potential memory conflicts that may arise can be used to estimate the possibility for parallel execution, since it reveals whether two PLEX jobs may be in conflict with each other through the access of the same data. We also think that such a case-study will give us ideas on the characteristics of the analysis that need to be specified, as well as the code transformations that need to be performed, i.e., it will give us the possible answers to Q2.

Due to the very high availability demands that exists for telephone exchange systems (which implies that system failures are costly), it is important that the analysis as well as the proposed transformations are safe. Therefore, the analysis and the transformations must be based on a formal semantics for

[2] In our case, there are approximately 20 Mlines of PLEX code in the AXE system.

PLEX.

This thesis will provide the necessary formal basis for the analysis and the transformations by specifying an operational semantics for PLEX. It will also report on a case-study of potential memory conflicts in some existing PLEX code.

Throughout this thesis, we will assume a conventional shared-memory architecture equipped with a run-time system that executes PLEX as it is (without any modification). The shared data is automatically protected through a locking scheme. The execution on this architecture is modeled in Chapter 4.4. Locking blocks will guarantee consistency of data, since data in a block can never be accessed by a PLEX job executing outside that block. However, it may be overly conservative, since two parallel PLEX jobs accessing the same block may well never touch the same data. This thesis aims at allowing a more loose locking scheme, where a block need not be locked if we know for sure that the PLEX jobs executing in it cannot have any memory conflicts.

1.3 Related Publications

The formal semantics for PLEX, as well as the study of potential shared-memory conflicts in the existing PLEX code, have been presented in the following publications:

- J. Lindhult, *A Structural Operational Semantics for PLEX*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-166/2004-1-SE, December 2003.

- J. Lindhult and B. Lisper, *A Formal Semantics for PLEX*. In Proceedings of the 2nd APPSEM II Workshop (APPSEM'04), Tallinn, Estonia, April 2004.

  Our first version of the operational semantics for sequential execution of PLEX was presented in the above technical report, and summarized in an Extended Abstract presented at APPSEM-04.

- J. Lindhult and B. Lisper, *Two Formal Semantics for PLEX*. In Proceedings of the 3rd APPSEM II Workshop (APPSEM'05), Frauenchiemsee, Germany, September 2005.

- J. Lindhult, *An Operational Semantics for the Execution of PLEX in a Shared Memory Architecture*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-227/2008-1-SE, April 2008.

  The first version of a semantics for PLEX in the shared-memory architecture was presented at APPSEM-05, and later refined in a following technical report.

- J. Lindhult and B. Lisper, *Sequential PLEX, and its Potential for Parallel Execution*. In Proceedings of the 13th International Workshop on Compilers for Parallel Computers (CPC 2007), Lisbon, Portugal, July 2007.

- J. Lindhult, *Existing PLEX Code, and its Suitability for Parallel Execution - A Case Study*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-228/2008-1-SE, April 2008.

  The initial results of our study of potential shared-memory conflicts in the existing PLEX code was presented at CPC 2007. The study was then completed in a technical report.

1.4 Contributions

The main contributions of this thesis are:

- By using a labeled program (following the style in [8]) we show a straightforward operational semantics for an imperative, non-toy like, language which includes the `GOTO` statement, and an asynchronous communication paradigm.

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

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

- A study of existing PLEX-code, and possible propositions on how the existing code, by a minimum of changes, could be transformed into suitable parallel code.

1.5 Thesis Outline

The thesis is structured in the following way: Chapter 2 contains an introduction to PLEX, and the AXE system. The parallel architecture, and different run-time systems are found in Chapter 3. The semantics for PLEX is specified in Chapter 4, whereas Chapter 5 covers examination of the potential memory conflicts. We discuss related work in Chapter 6, before we conclude, and discuss future work, in Chapter 7.

Chapter 2

AXE and PLEX

We will start this chapter with a brief description of the AXE telephone exchange system, followed by an introduction to the language PLEX. For a more thorough description, we refer to [9].

2.1 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 structure of the main parts of the system is shown in Fig 2.1.

The part of the system that is in focus for parallel processing is the Central Processor Sub-system, which architecture is shown in Fig. 2.2. In the current architecture, 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 requests are received by the RP's, and processed by the CP;

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.

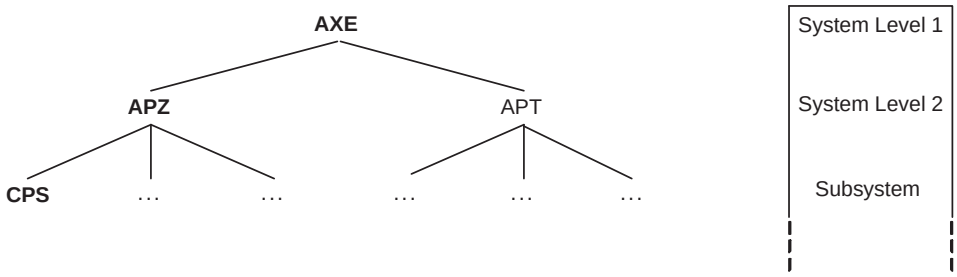

Figure 2.1: *The (original) hierarchical structure of the AXE system.*

2.2 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. 2.2. 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 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 subprograms together with block-wise local data, see Fig. 2.3. As we will see in Section 2.3, this data (variables) can be classified into different categories depending on whether or not the value of a variable 'survives' termination of the software. 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

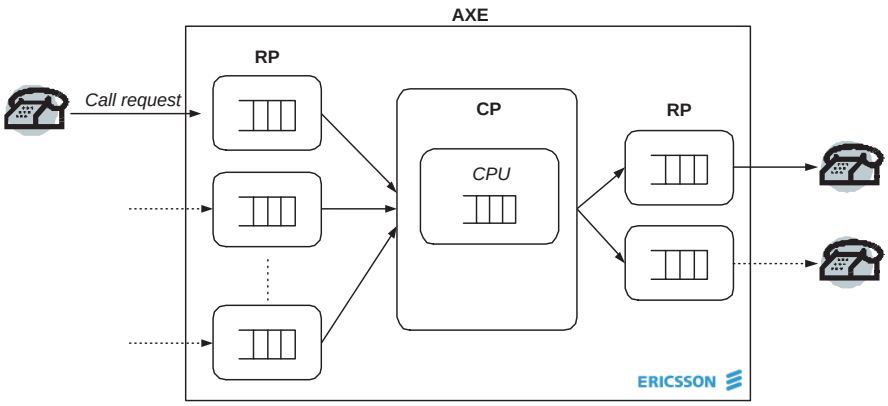

Figure 2.2: *Current (single-processor) architecture of the Central Processor Sub-system.*

blocks is strictly hidden, and there is no other way to access it than through the sub-programs.

The sub-programs in a block can be executed in any order: execution of a sub-program is triggered by a certain kind of event called *signal* arriving to the block. Signals may be *external*: arriving from the outside or *internal*: arriving from other sub-programs, possibly executing in other blocks. The execution of one, or several, sub-programs constitutes a *job*; a job begins with a signal receiving statement, and is terminated by the execution of an `EXIT` statement. Due to the 'atomic' execution of a job, i.e., once a job is started it will run to completion, we may also view them as a kind of transactions.

With *job-tree*, we denote the set of jobs originating from the same external signal. See also Fig. 2.4 (b), where the corresponding job-tree for the execution in Fig. 2.4 (a) is shown.

Since sub-programs can be independently triggered, it is accurate to consider jobs as "parallel". 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. 2.6 (b) and 2.4 (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

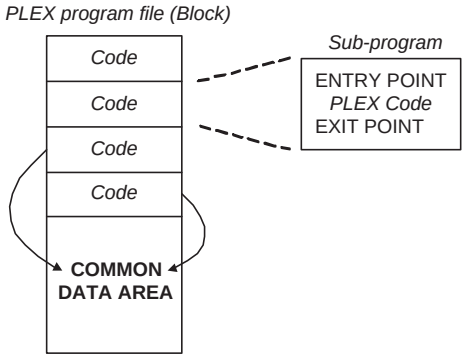

Figure 2.3: *A PLEX program file (a* block*) consists of several sub-programs.*

executed non-preemptively. 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).

2.3 Shared Data

Since the data in a block is shared between all its sub-programs, it might seem as **all** variables may be potentially shared. However, as we indicated in Section 2.2, the variables belong to different categories: basically, the variables can be divided into the following two main categories; *data stored (DS)* or *temporary*.

- The value of a temporary variable exists only in the internal processor registers, and only while its corresponding software is being executed. Variables are by default temporary, and thus cannot be shared between different jobs.
- DS variables are persistent: they are loaded into a processor register from the memory when needed, and then written back to the memory. These variables can be further divided into[1]:

  1. *Files*

[1] See also [10] where this distinction is discussed more thoroughly.

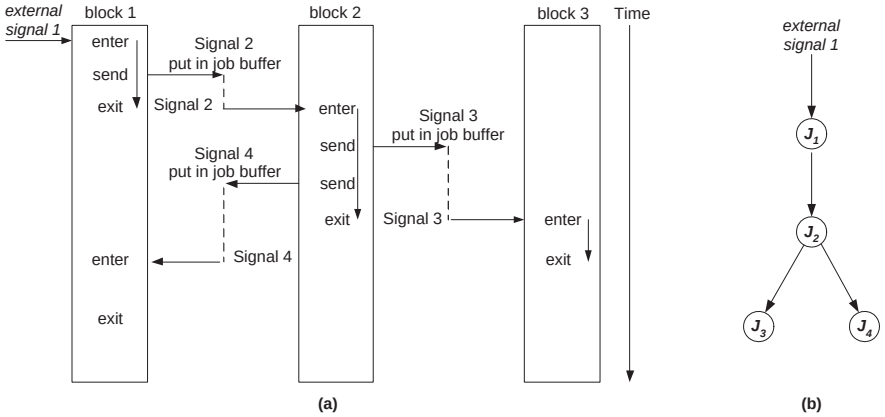

Figure 2.4: The ”pseudo-parallel” execution model of PLEX (a), and a corresponding job-tree (b).

2. *Common variables*

Common variables are (mostly) “scalar” variables (but may as well be arrays), whereas files essentially are arrays of *records* (similar to “structs” in C). Elements of records are called *individual variables*. *Pointers* address the relevant record in a file. The records in a file are numbered, and the value of the pointer is the number of the current record. Notable is that a pointer ”behaves” like a temporary variable in that it will lose its value when the job that uses the pointer terminates. Thus, common variables are used to store the ”current value” of a pointer between the execution of different jobs.

PLEX	**C**
record	struct
file	array of structs
pointer	array index
individual variable	struct member
common variable	global variable

Table 2.1: *Some PLEX concepts, and their ”counterparts” in C.*

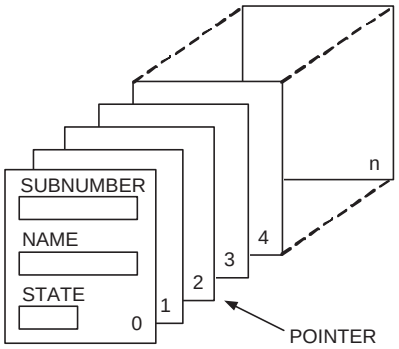

Figure 2.5: *An example file with n records and a pointer with the current value 2.*

Fig. 2.5 shows an example file with its records and a pointer, whereas Table 2.1 tries to relate the above PLEX concepts to its closest counterpart in C.

2.4 Signals

A key aspect, which distinguishes PLEX from an "ordinary" imperative language, is the asynchronous communication paradigm: jobs communicate and control other jobs through signals.

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 through combinations of different properties, where the main distinction (from a semantical point of view) is between *direct* and *buffered* signals, see Fig. 2.6. 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 (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. 2.6. 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 $job–tree$.

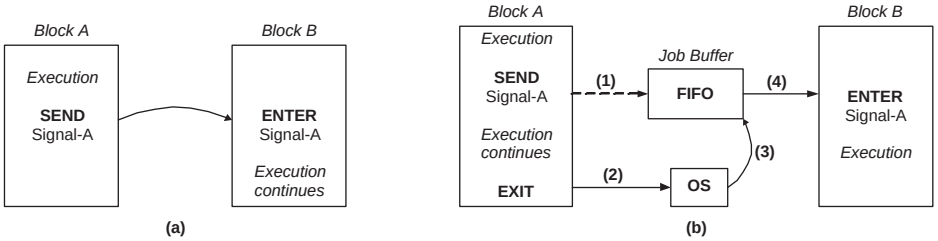

Figure 2.6: (a): a direct signal, ”similar” to a jump. (b): buffered signals: a buffered signal is sent from Block A which is inserted at the end of the job buffer (1). When the job in Block A terminates, the control is transferred to the OS (2), which fetches a new signal from the buffer (3). This signal then triggers the execution in 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. 2.7. The combined signal is always direct, while the single signal may be buffered.

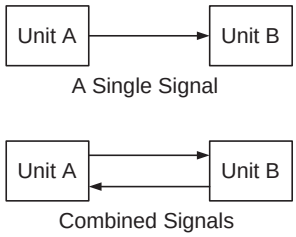

Figure 2.7: *Single and combined signals.*

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. 2.2.

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.

2.5 Application Modules, and the Resource Module Platform

The AXE *Source System* is a number of hardware **and** software resources developed to perform specific functions according to the customer'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. 2.8, where it is also shown that different AMs can be used in more than one application. The related pieces of software, mentioned above, are the PLEX blocks (Section 2.2), 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

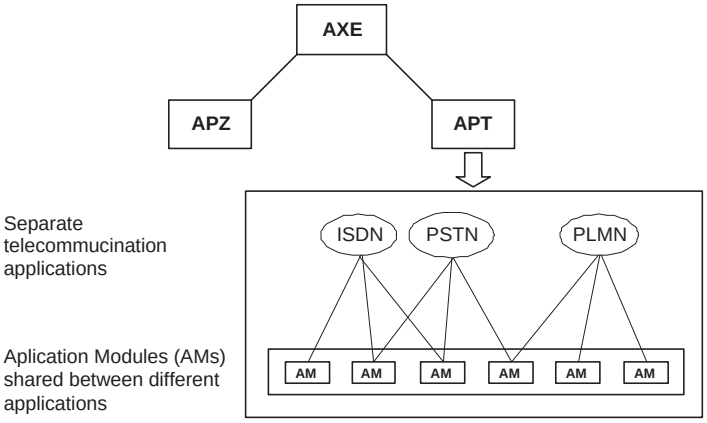

Figure 2.8: *The AM concept incorporated into the AXE system.*

required exchange was constructed by combining the necessary AMs into an exchange with the required functionality (i.e., with the necessary applications).

An *AM based system*, consists of the AMs (which forms the applications) together with some common resources. The common resources are collected in the *Resource Module Platform*, or RMP for short. As can be seen in Fig. 2.9, communication between different AMs is performed via an AM protocol, whereas communication between an AM and the RMP is performed via ordinary signals (as described in Section 2.4).

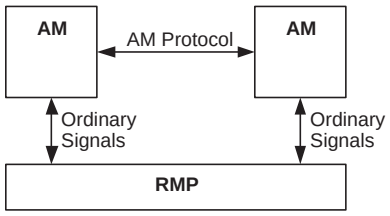

Figure 2.9: *Illustration of An AM based system (from [11]).*

Chapter 3

Execution Paradigms

As we said in Chapter 1.2, the parallel semantics in Chapter 4.4 models the execution of PLEX on a conventional shared-memory architecture. The architecture is assumed to be equipped with a run-time system, which is designed to execute PLEX programs as they are, i.e., unmodified. The run-time system, *CMX-FD*, is covered in Section 3.3, and its forerunners, *FD*, and *CMX*, are covered in Section 3.1 and Section 3.2, respectively. Common for these run-time systems (or execution paradigms) are that all require a shared memory architecture with support for Thread-Level Parallelism (TLP) as shown in Fig. 3.1. Examples of such architectures are Symmetric Multiprocessors (SMP), Chip-Multiprocessors (CMP), and Simultaneous Multi-Threading processors (SMT).

Common for the execution paradigms considered is that the old (sequential) software, without modifications, would be executed on the parallel architecture. The run-time systems are designed to preserve functional equivalence with the original, sequential system. The approach taken to achieve this equivalence is to (1) let jobs from the same job-tree execute in the same sequential order as in the single-processor case, and (2) lock a block as soon as a job is executing in it in order to protect its data from being concurrently accessed.

Although the use of a locking scheme introduces the risk of deadlocks, we will not consider this further since the run-time systems are assumed to have a mechanism to resolve this.

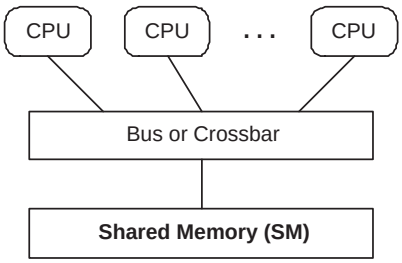

Figure 3.1: *A conventional, shared memory, multi-processor, architecture.*

3.1 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 only exists 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. 3.2).

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.

3.2 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. 3.3). 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, locking 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.

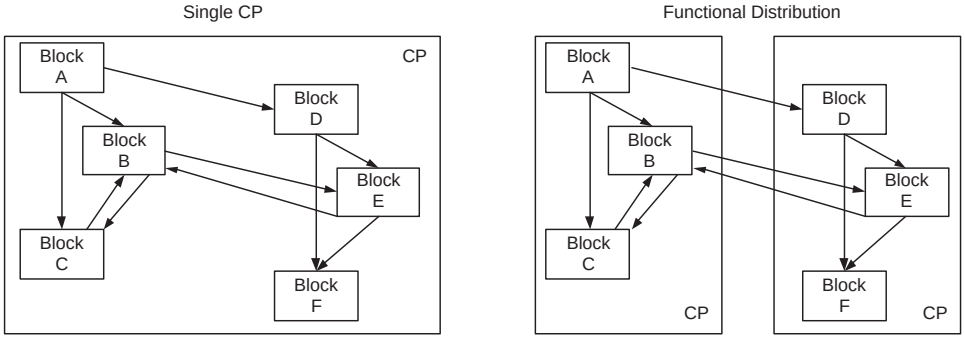

Figure 3.2: *Example (from [11]) on the FD principles: blocks, that in the single-pro. case is executed on the same CP, is in FD distributed over the available resources.*

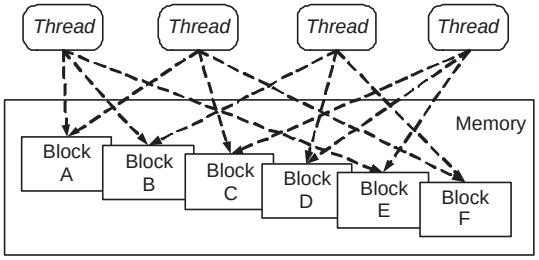

Figure 3.3: *The CMX paradigm, where any of the threads can execute any of the blocks that resides in memory.*

3.3 CMX-FD

The assumed execution model for parallel execution of PLEX, and the one modeled in Chapter 4.4, is CMX-FD. As hinted by the name, CMX-FD is the combination of the previous described execution paradigms CMX and FD. 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 Chapter 2.5).

Now that we have discussed both Functional Distribution (FD), Section 3.1, and CMX, Section 3.2, 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[1] 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 (Chapter 4.4.2), it must be possible for any thread to execute these blocks, which implies the shared memory. It should be stated that the CMX-mode 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. 3.4, 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, 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 Chapter 4.4.2 when we specify the parallel semantics for buffered signals.

[1] These direct signals are in almost every case combined signals. (The different kind of signals was discussed in Chapter 2.4.)

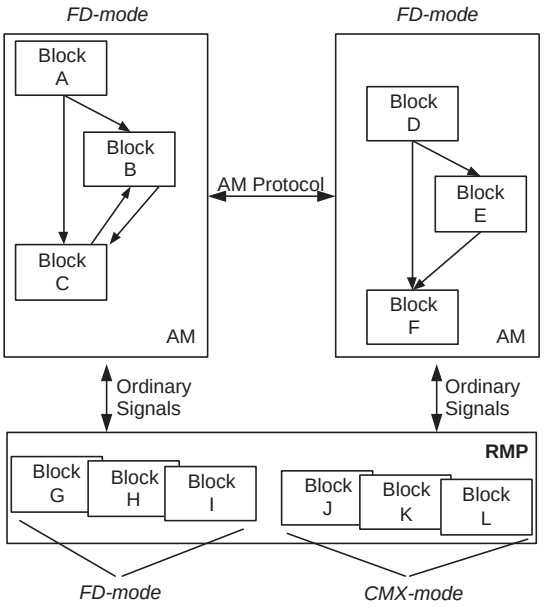

Figure 3.4: *Illustration of the CMX-FD execution paradigm.*

Chapter 4

Operational Semantics for Core PLEX

Until recently, the semantics for PLEX has been defined through its implementation, but in the following sections we will present an operational semantics for the current single-processor architecture, as well as for the multi-threaded shared-memory architecture described in Chapter 3. The semantics, given in terms of state transitions, will be specified for the language *Core PLEX*.

The chapter starts with an introduction to programming language semantics, intended for the reader not familiar with the subject. Section 4.2 defines our modeled language Core PLEX. The sequential, and the parallel semantics for Core PLEX is presented in Section 4.3, and Section 4.4 respectively.

4.1 Programming Language Semantics

Programming language semantics is concerned with rigorously specifying the meaning, or *the effect*, of programs that are to be executed. By effect we mean, for instance, the contents of the memory locations, which parts of the program that are to be executed, or the behavior of the hardware affected by the program. A semantic specification captures these things in a formal way.

Formal descriptions using grammars of programming languages are becoming more and more popular, e.g. the BNF[1] is used to specify the syntax

[1] Backus-Naur Form

of PLEX. The problem is that a formal description of the syntax says nothing about the meaning of the program since "*syntax is concerned with the grammatical structure of programs*" whereas "*semantics is concerned with the meaning of grammatically correct programs*" [12] (both quotations). Or put in other words, since a formal syntax only tells us which sequences of symbols that forms a legal program, it is not enough if we need to reason about the meaning of program execution. To be able to do that, we need the formal semantics.

Typical uses for a semantic specification of a language is

- to reveal ambiguities and complexities in what may look as a clear documentation of the language (e.g. the language manual).
- to form the basis for implementation, analysis and verification.

4.1.1 Semantic Approaches

The meaning of a programming language can be formalized in different ways. In the general case, the semantics will tell us something about the relation between an initial and a final state. Standard literature, for instance [12], normally classifies a semantic approach as one of the following three categories:

- *Operational semantics - How to execute the program*. An operational approach is not only concerned with the relationship between the initial and the final state, it also reveals *how* the effect of the computations is produced. The meaning is often specified by a transition system (or sometimes as an abstract machine). The different operational approaches differ in the level of details:
  - In the *Natural/Big steps semantics*, the focus is on how the overall results of the executions are obtained. The transition system specifies this relationship for every statement and is usually written in the form $\langle S, s\rangle \rightarrow s'$ which, intuitively, means that the execution of the statement S from state s will terminate and the resulting state will be s'.
  - As opposite to the big steps semantics, the *Structural operational /Small steps semantics*, is also concerned with how the individual steps of the execution takes place. If the execution of the statement S in state s leads to the final state s', the small steps semantics

will tell us something about the intermediate states that are "visited" during the execution. In other words, the focus is on the individual steps of the execution. The transition system has the form $\langle S, s\rangle \Rightarrow \gamma$ where γ is *either* of the form $\langle S', s'\rangle$ **or** of the form s'. This means that the transition system expresses the **first** step of the execution of the statement S from the state s and the result of this is γ.

- *Denotational semantics - The effect of executing the program*. Denotational semantics, in contrast to operational semantics, is only concerned with the effect of the computation, not how it is obtained. Meanings are modeled by mathematical objects (usually functions) representing the effect of executing the constructs. The meaning of sequences of statements are then modeled by function composition like $f \circ g$

- *Axiomatic*[2] *semantics - Partial correctness properties of the program*. The axiomatic approach is concentrated on specific properties of a program. These properties are expressed as *assertions*. Axiomatic semantics involves rules for checking these assertions. There may be aspects of the executions that are ignored since only specific properties are considered. Axiomatic definitions is often given in the form $\{P\}S\{Q\}$ where P is a Pre-condition, S the statement to be executed and Q a Post-condition. This is to be interpreted as: "*If P holds **and** the execution of S terminates, **then** Q will hold*".

4.2 Core PLEX

As we said in the beginning of this chapter, the semantics for PLEX will be given in terms of a semantics for the language Core PLEX. Core PLEX 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.

Notable omissions from the real PLEX language are the statements for signal reception (see below), and statements for iteration and selection (`CASE`). Although simplified, it is actually possible to express many of the omitted

[2]The axiomatic approach is usually referred to as Hoare logic after the original paper [13]

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	::=	$a_1\ op_r\ a_2$
$data$	::=	$\{\textbf{Var}\vert\textbf{Num}\}^k \perp^{25-k}, \quad 1 \leq k \leq 25$
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 4.1: The abstract syntax for Core PLEX.

PLEX statements in terms of already specified Core PLEX statements (as we will do in Section 4.3.5). We may therefore view the modeled language as the "Core" of PLEX.

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 4.1. Following [8], 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 $Stmt$: $\textbf{Lab} \rightarrow \textbf{Stmt} \cup \textbf{BExp}$ 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 $Stmt^{-1}$: $\textbf{Stmt} \cup \textbf{BExp} \rightarrow \textbf{Lab}$

In Chapter 2.2, 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 $\mathbf{ELab}$ into three disjoint sets $\mathbf{Dir}$, $\mathbf{Buf}$, $\mathbf{Ext}$ for the respective labels. Furthermore, we partition $\mathbf{Buf}$ into the disjoint sets $\mathbf{LevA}$, and $\mathbf{LevB}$, in order to capture the different priorities among the signals. (Recall from Chapter 2.4 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 [8], through the three functions $init\colon \mathbf{Stmt} \to \mathbf{Lab}$, $final\colon \mathbf{Stmt} \to P(\mathbf{Lab})$, and $Flow\colon \mathbf{Stmt} \to P(\mathbf{Lab} \times \mathbf{Lab})$ in Table 4.2. 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 $succ$, $succT$, $succF\colon \mathbf{Lab} \to \mathbf{Lab}$ 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:*

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

Further on, we recall that the code (and the data) is structured in blocks (Chapter 2.2), and we assume that the program under consideration consists of

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	$final(S_1) \cup final(S_2)$	$flow(S_1) \cup flow(S_2) \cup$ $\{(l, init(S_1)), (l, init(S_2))\}$
$[\texttt{SEND}\ signal]^l$ $(signal \in \mathbf{Dir})$	l	$\emptyset$	$(l,\ signal)$
$[\texttt{SEND}\ signal]^l$ $(signal \in \mathbf{Buf})$	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 4.2: Definition of $init$, $final$, and $Flow$. Note that since it is irrelevant for the definitions of the above functions whether or not a signal carries any data, we have omitted those cases from the table above.

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 $\mathbf{Var}$ and $\mathbf{Lab}$, 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 Chapter 3.3 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 4.4.2, when specifying the parallel semantics for the signal statements.)

4.3 A Sequential Semantics

Since the execution of statements are modeled as state transitions, we begin this section by defining the state of the system. States are modeled by tuples of the form

$$\begin{aligned} s \;&=\; \langle \mathcal{VSC}, JBA, JBB, \sigma, \delta \rangle \\ &\in \mathbf{Lab} \times [(\mathbf{ELab}, data)] \times [(\mathbf{ELab}, data)] \times (\mathbf{Var} \to N) \times [\mathbf{Lab}] \end{aligned}$$

We continue by 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 system 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 4.1, 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 $\sigma|_{RM} \mapsto data$ 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

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

- 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 Chapter 2.4, and Fig. 2.7). We therefore add the context information δ to the 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 transition rules from state to state. The transition relation $\rightarrow$ specifies how the statements are executed. The transitions have the form

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

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 4.3.4). 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 candidates for parallel execution (Section 4.4).

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

$$s = \langle \perp, \varepsilon, \varepsilon, \sigma|_{RM} \mapsto \emptyset|_{DS} \mapsto \Upsilon, \varepsilon \rangle$$

The initial state expresses that the $\mathcal{VSC}_i$ does not map to any statement; the temporary storage (RM) is empty; the are no signals in the JBA or JBB job-buffer (which we recall is modeled as lists of signals). 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 [9] where this is discussed in more detail) more than to say that the variables in the DS always have some initial values Υ. δ contains an empty value since no job has been started yet, and consequently there is no context information available.

4.3.1 The Basic Statements

Starting in this section, we will specify the semantics for Core PLEX in the current, single-processor architecture. We begin with what we call the *basic*

statements, i.e., assignments[3], jump-statements, conditionals, and iterations, and we continue with the semantics for the signal statements in Section 4.3.2[4].

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{x:=a}$$
$$\langle succ(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma],\ \delta\rangle$$

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

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2}$$
$$\langle succT(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma,\ \delta\rangle$$
$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{tt}$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2}$$
$$\langle succF(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma,\ \delta\rangle$$
$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{ff}$$

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

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{GOTO}\ label}$$
$$\langle label,\ JBA,\ JBB,\ \sigma,\ \delta\rangle$$

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

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

[3] 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).

[4] The transition rules for the sequential semantics are summarized in Appendix A.

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!

4.3.2 The Signal Statements

Before we continue with the semantics for the different signal statements, we recall (from Section 4.2) 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, $\mathbf{ELab}$ has been partitioned into the disjoint sets $\mathbf{Dir}$, $\mathbf{Buf}$, $\mathbf{Ext}$ in order to distinguish between direct, buffered, and external signals. $\mathbf{Buf}$ has then been partitioned into the disjoint sets $\mathbf{LevA}$, and $\mathbf{LevB}$, in order to capture the different priorities among the signals.

We begin this part with the statements for the single[5] signals

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta \rangle \xrightarrow{\texttt{SEND}\ signal} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto \bot,\ \delta \rangle \quad \text{if} \quad signal \in \mathbf{Dir}$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta \rangle \quad \text{if} \quad signal \in \mathbf{Dir}$$

The following rules deal with the sending of a buffered signal. The first two cases deals with the sending of a priority A signal, whereas the last two handles signals of priority B.

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta \rangle \xrightarrow{\texttt{SEND}\ signal} \langle succ(\mathcal{VSC}),\ JBA : (signal, \bot),\ JBB,\ \sigma,\ \delta \rangle \quad \text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevA}$$

[5]The single signals do not, in contrast to the combined signals, require a reply. For a discussion about the different signal properties, see Chapter 2.4.

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data}$$
$$\langle succ(\mathcal{VSC}),\ JBA : (signal, data),\ JBB,\ \sigma,\ \delta\rangle$$
$$\text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevA}$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal}$$
$$\langle succ(\mathcal{VSC}),\ JBA,\ JBB : (signal, \perp),\ \sigma,\ \delta\rangle$$
$$\text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevB}$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data}$$
$$\langle succ(\mathcal{VSC}),\ JBA,\ JBB : (signal, data),\ \sigma,\ \delta\rangle$$
$$\text{if} \quad signal \in \mathbf{Buf},\ \ signal \in \mathbf{LevB}$$

The concept of combined signals is shown in Fig. 4.1, and we recall from Chapter 2.4 that the difference between a combined signal and other direct signals[6] is that the combined signal always requires an answer (a reply signal).

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

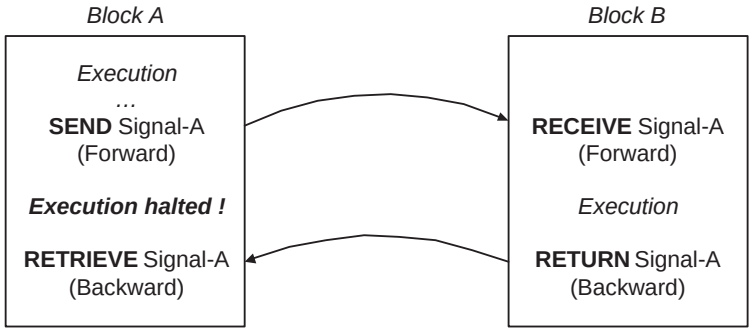

Figure 4.1: *The PLEX statements for sending/receiving combined signals. Note that the signal receiving statements is omitted in Core PLEX (see Chapter 4.2).*

The semantics for the combined signals are as follows

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label} \langle cfsig,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto \bot,\ label : \delta\rangle$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label} \langle cfsig,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data,\ label : \delta\rangle$$

$$\langle \mathcal{VSC},\ JBA,\ ,JBB,\ \sigma,\ label : \delta\rangle \xrightarrow{\texttt{RETURN}\ cbsig} \langle label,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto \bot,\ \delta\rangle$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ label : \delta\rangle \xrightarrow{\texttt{RETURN}\ cbsig\ \texttt{WITH}\ data} \langle label,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data,\ \delta\rangle$$

We end this section with the semantics for the local signals, and as was said in Chapter 2.4, 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).

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{TRANSFER}\ signal} \langle signal,\ JBA,\ JBB,\ \sigma,\ \delta\rangle$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{TRANSFER}\ signal\ \texttt{WITH}\ data} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data \setminus \bot,\ \delta\rangle$$

4.3.3 The `EXIT` Statement

We recall from Chapter 2.2, that the `EXIT` statement marks the termination of an ongoing job. At termination, a new job is immediately started as the control is transferred to the first signal label in the job queue. However, a job of priority `B` is only allowed to be started if there isn't any job of priority `A` waiting to be executed. This motivates the two different `EXIT`-transitions.

$$\langle \mathcal{VSC},\ (signal, data) : JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{EXIT}} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta\rangle$$

$$\langle \mathcal{VSC},\ \bot,\ (signal, data) : JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{EXIT}} \langle signal,\ \bot,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta\rangle$$

4.3.4 Additional transitions

The following 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. The rules are always enabled, and they introduce nondeterminism into the semantics:

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\epsilon} \langle \mathcal{VSC},\ JBA : (signal, data),\ JBB,\ \sigma,\ \delta\rangle \quad \text{if } signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevA}$$

$$\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\epsilon} \langle \mathcal{VSC},\ JBA,\ JBB : (signal, data),\ \sigma,\ \delta\rangle \quad \text{if } signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevB}$$

4.3.5 Translating Selection, and Iterations Statements into Core PLEX

As said in Section 4.2, the PLEX statements for iteration and selection are omitted in Core PLEX. We therefore conclude the sequential semantics part with translating the omitted statements into equivalent Core PLEX statements.

The statement for selection (CASE) is in many ways similar to the SWITCH statement in C. The CASE statement has the general form

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

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 already specified conditional GOTO-, and the shortened IF-statements (Section 4.3.1), we can express the CASE statement in terms of the IF−THEN−ELSE statement in the following way;

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

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

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

Next, we look at the different iteration statements that are available in PLEX. From [14], 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

[7]The AXE system has been classified as a soft real-time system by Arnström et. al in [15].

ON $pointer/variable$ FROM $expression_1$ UPTO $expression_2$ DO S

ON $pointer/variable$ FROM $expression_1$ DOWNTO $expression_2$ 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

$$
\begin{array}{ll}
 & i = expression_1 \\
LFalse\ ) & \texttt{IF}\ i = expression_2\ \texttt{THEN GOTO}\ LTrue \\
 & S \\
 & i = i{+}1 \\
 & \texttt{GOTO}\ LFalse \\
LTrue\ ) & remaining\ statements
\end{array}
$$

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;

$$
\begin{array}{ll}
 & i = expression_1 \\
LFalse\ ) & \texttt{IF}\ i = expression_2\ \texttt{THEN GOTO}\ LTrue \\
 & S \\
 & i = i{-}1 \\
 & \texttt{GOTO}\ LFalse \\
LTrue\ ) & remaining\ statements
\end{array}
$$

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.

[8] See for instance [16]

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
$LDone$) $remaining\ statements$

4.4 A Parallel Semantics

The parallel semantics in this thesis models the execution of PLEX on the architecture and run-time system described in Chapter 3. Logically, the execution is done by a static number of threads, which may or may not equal the number of processors. Each thread has its own local state and a number of pre-allocated blocks, which are only executed by the thread they have been allocated to. The "remaining" blocks can be executed by any of the threads, and we say that these blocks execute in "parallel mode".

Similar to the sequential semantics (in Section 4.3), the parallel semantics is given in terms of state transitions, but whereas the sequential semantics only needed to consider 'one' state;

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

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 is 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 components in the above state are explained below.

- $\mathcal{VSC}_i$ now holds the local program counter for thread T_i.
- JBx, where $x = \{A,\ B\}$ are sequences of entry (signal) labels as defined in Section 4.3.
- Since the code we consider may be accessed by any of the k threads, the variables in the system, σ, are now found in the global part of the state, s_G. The division

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

  made in Section 4.3 is still valid. However, the fact that some variables (RM) are only used for temporary storage of data that are local to a job implies that RM can be divided into the following disjunct sets

  $$RM \ = \ RM_1, \ \ldots, \ RM_k \ \text{ such that } \ RM_i \cap RM_j \ = \ \emptyset \text{ for any } i \neq j$$

  to capture the temporary variables used by the job executing at thread T_i.

  The notation $\sigma|_{RM} \ \mapsto \ data$ has already been covered in Section 4.3. Here, we only note that we will now write $\sigma|_{RM_i} \mapsto data$ to denote

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

- In Chapter 3, we said that the parallel run-time system 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 $\mathbf{LVar}$, which is a set of β binary *lock variables* $L_1, \ldots, L_\beta$, distinct from any variables in $\mathbf{Var}$. 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 (in Chapter 3) 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 the first transition in Section 4.4.4). The creation of a job-tree is captured in Section 4.4.4 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

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

- The context information δ is now associated with thread T_i, and is correspondingly indexed δ_i.

For each parallel rule we omit the 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[9] will now have the form

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

for a transition that affects the local memory of thread T_i, and where $Stmt(\mathcal{VSC}_i) = S$. Similar to the sequential semantics, there are also some transitions labeled with ϵ; the rules modeling the arrival of an external signal, as well as the rule for starting a new job[10] are found in Section 4.4.4.

As in Section 4.3, we will only consider the general case; execution on the Traffic handling level (priority `B`). The reason is that these are the jobs that would be executed in parallel.

The initial start up state would have the following contents:

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

This 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; 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). $\mathcal{F}_i$ 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.

4.4.1 The Basic Statements

The parallel transition rules for the basic statements are straightforward "parallelizations" of the similar rules for the sequential semantics in Section 4.3.1, and they should not need any further explanation[11].

If nothing else is stated, the transitions in the following sections 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).

[9] Similar to Section 4.3, $\rightarrow$ defines the transition relation.

[10] Unlike the sequential semantics, the transition for the `EXIT` statement does **not** start a new job, only terminates the current one. This is further discussed in Section 4.4.3.

[11] The parallel semantics is summarized in Appendix B.

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

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i}$$
$$\langle succT(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$
$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{tt}$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i}$$
$$\langle succF(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$
$$\textbf{if}\ \ \mathcal{B}[\![b]\!]\sigma = \texttt{ff}$$

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

4.4.2 The Signal Statements

Before we continue with the semantics for the different signal statements, we need to cover the receiver of a specific signal (i.e., the receiving thread). For direct, and buffered signals[12]the following applies:

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 4.4), 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.

[12]We recall (Chapter 2.4) 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 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 Chapter 3.3), 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 4.3, where we also see when information about the *Home thread* for a given CMX-mode block (which we discussed in Chapter 3.3) is of importance.

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 4.3: *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

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

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_is 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 Chapter 3) 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 4.4.3, 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[13] signals

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i}$$
$$\langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \quad signal \in \mathbf{Dir},\ \gamma = LB(signal),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i}$$
$$\langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \quad signal \in \mathbf{Dir},\ \gamma = LB(signal),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

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 at the end of this section.

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

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

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

[13] The single signals do not, in contrast to the combined signals, require a reply. For a discussion about the different signal properties, see Chapter 2.4.

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

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

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

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

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

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

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

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

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

The concept of combined signals was shown in Fig. 4.1, and we recall from Chapter 2.4 that the difference between a combined signal and other direct signals[14] is that the combined signal always requires an answer (a reply signal).

The semantics for the combined signals are as follows

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

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

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

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

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$
$$\xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label,\ i}$$
$$\langle cfsig, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \quad \gamma = LB(cfsig),\ (\sigma_L(L_\gamma) = 0 \vee L_\gamma \in Locks_i)$$

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

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

The transition rules for the local signals are straightforward parallel versions of corresponding sequential rules.

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

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

In conjunction with the semantics for the sending of a buffered signal, we mentioned (on page 48) 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).

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

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

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

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

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

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

4.4.3 The `EXIT` Statement

When the `EXIT` statement was covered in the sequential semantics (Section 4.3.3), we noted that the transition specified the termination of the current job, as well as the start of a new job. This is not the case for its parallel counterpart (as we indicated in Section 4.4.2, on page 47). 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 only 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.

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

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

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

$$\langle \bot, JBB_i, \emptyset,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle \rangle$$
$$\text{if} \quad Receiver(signal) = i$$

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

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

4.4.4 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 4.4.)

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

$$\langle signal, JBB_i, \{L_\gamma\},\ [\mathcal{T}] : \mathcal{F}_i\ -\ [signal : \mathcal{T}],\ \delta_i,$$
$$\langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$
$$\text{if} \ \ \gamma = LB(signal),\ \sigma_L(L_\gamma) = 0,\ s_1(JBA) = \varepsilon$$

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:

$$\langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$$
$$\langle \mathcal{VSC}_1,\ JBA : (signal, data),\ JBB_1,\ Locks_1,\ \mathcal{F}_1 : [signal],\ \delta_1,\ \langle \sigma,\ \sigma_L \rangle \rangle$$
$$\text{if } signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevA}$$

$$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$$
$$\langle \mathcal{VSC}_i,\ JBB_i : (signal, data),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$
$$\text{if } signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevB}$$

4.4.5 Global Transitions

In Section 4.4.1 - 4.4.4, we have defined the meaning of each individual Core PLEX statement, when executed 'locally' by thread T_i. However, the description is not complete as long as we don't consider the concurrently executing threads.

We recall (from Section 4.4) 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[15]

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

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

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).

[15] The transitions are of standard form as used for instance in [17]

Chapter 5

Case Study: Examining Potential Memory Conflicts

In order to estimate the possibility for parallel execution of the existing PLEX code, we have performed a static program analysis of the potential memory conflicts that actually can arise. A second motivation for this case-study was to get ideas on the characteristics of the analysis that needs to be specified (as discussed in Chapter 1.2).

The number of conflicts are measured as the relative numbers of different signals that can interfere with each other through the shared data areas. Initially, we assumed the worst case scenario; i.e., a conflict between every pair of examined signals. This assumption corresponds to a 100% conflict rate between the examined signals. However, our results show that compared to a straightforward parallel implementation, where each shared data area is protected by a lock, we can by a simple static analysis of the data usage reduce the potential conflicts between signals to be in the range 0-76% for the observed signals, thereby reducing the amount of manual work that probably still needs to be performed in order to adapt the code for parallel processing.

5.1 Analysis of Conflicts

We say that two signals in the same block are in *conflict* if they might access the same variable in such a way that the consistency of data is threatened if the code triggered by the signals is executed concurrently. This is the case if both

signals might access the variable and at least one may write to it. If two signals are *not* in conflict, they may safely be executed concurrently with no protection at all. A run-time system may use this information to lock a block selectively only for signals that are in conflict. This improves on the parallel architecture in Chapter 3, which locks a block as soon as one of its signals is executed.

To determine whether or not two signals might be in conflict with each other, the usage of each variable in each signal is classified in the following way:

- $\bot$ - The variable is **never** used by the signal in question.
- R - Read Only, i.e., the only way the signal is accessing the variable is in read operations.
- W - If the signal accesses the variable, the first access will **always** be a write operation.
- $\top$ - It is not possible to (statically) classify the variable according to the previous cases, i.e., the usage of the variable might be input dependent, or there might be different paths through the code that use the variable in different ways. It might also be the case that the signal performs a read operation as a first access to the variable.

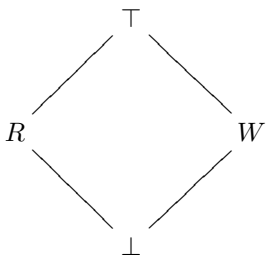

Figure 5.1: *The hierarchical ordering of variable usage.*

Based on our knowledge on how the variables are used, we can order them in a hierarchical way as in Fig. 5.1, where we go from absolute knowledge ($\bot$- never used) to actually no knowledge at all ($\top$- can't always be determined). We also make the following observations:

- A variable that is never used ($\bot$) can never cause the signal to be in conflict with other signals.

- The value of a Read Only variable is only used (read from), and similar to $\perp$ does not cause the signal to be in conflict with other signals unless some other signal writes to the variable.
- For a variable classified as W, we notice that if every signal that accesses the variable always performs a write as a first possible access, it will be safe to perform the following transformation; let each signal work on a local copy of the variable. This does not change the semantics of the program since no signal will ever use a value written by another signal, regardless of whether or not this transformation is performed. We denote this transformation as 'w-optimization' in the following sections.
- Since an unambiguous use of a variable classified as $\top$ can not be determined, we must **always** assume a potential conflict between signals that use this variable.

A conflict matrix for each block would then be straight forward to deduce based on the classification of variables. We give a small example; consider three signals $Sig_{1,\,2,\,3}$, and three variables $Var_{1,\,2,\,3}$. Table 5.1 shows how the signals use the variables, as well as the corresponding conflict matrix. The conflict matrix indicates potential conflicts between Sig_1 and Sig_2, and between Sig_1 and Sig_3. Sig_2 and Sig_3 can however execute concurrently.

	Var_1	Var_2	Var_3
Sig_1	R	W	$\perp$
Sig_2	R	R	R
Sig_3	$\perp$	R	R

	Sig_1	Sig_2	Sig_3
Sig_1		X	X
Sig_2	X		
Sig_3	X		

Table 5.1: *Variable usage in three example signals, together with a corresponding conflict matrix.*

Once the conflict matrix has been constructed, it could for instance be used by the run-time system to perform a table look-up before allowing a signal to start executing.

5.2 Examining the Code

As we said in the previous section, only a subset of the blocks are executed in "parallel mode" which means that they may be executed by any thread,

and consequently are candidates for parallel execution. Other blocks are not considered in this study. Furthermore, every block might also, besides ”call processing code” (i.e., handling of telephone calls), contain some administrative code (e.g., charging for a call). Since administrative code is not allowed to execute while there is call processing work to be done, the variables that are considered in this study are DS variables accessed by call processing code (and where the block executes in ”parallel mode”).

Our studies are performed on existing PLEX code, for which we assume the shared-memory architecture described in Chapter 3 (and with the CMX-FD run-time system, Chapter 3.3). The software consists of 1045 blocks; 34 of those are executed in ”parallel mode” whereas the remaining blocks are allocated to different threads. A total of four blocks have been examined. Common for these blocks is that their (fraction of the) execution time is known to be high compared with other blocks. Each examined block contains a number of signals that are known to be executed significantly more frequently than other signals in each respective block. We call these “HF-signals” (High Frequency Signals). For every *DS variable* that are read from, or written to, by such a HF signal, we have examined the usage of this variable in every other signal in that block in order to find out which signals that may possibly be in conflict with these HF-signals. Table 5.2 summarizes the characteristics of each examined block: type of block, fraction of execution time, as well as the number examined signals, and variables.

The code has been inspected manually, and the reason for not trying to automate the process was that we believed that manual inspection also would increase our general knowledge on how the language is used, in “reality”, i.e., it would be ”*possible to “see” the semantics of the program*” [10].

Block	**Type**	**HF**	**Examined signals**	**Examined variables**
CHVIEW	*Middleware*	6	70 of 92	26
LAD	*OS*	2	8 of 28	88
MFM	*OS*	3	26 of 75	53
MSCCO	*Application*	2	68 of 75	16

Table 5.2: *The examined blocks.*

As we pointed out in Chapter 2.3, the persistent data are divided into Files and Common variables, where Files are arrays of records, and Common variables in most cases are ”scalar” variables. For Files, we make the following observation:

a conflict takes place through a file only when the same individual variable, in the same record, is accessed simultaneously. For a file of size n the probability of two accesses going to the same record is $1/n$, if the accesses are random, independent, and equally distributed. Files in PLEX are usually used to hold subscriber data and/or data generated during a telephone call [10]. The index of an access thus usually depends on externally supplied data, like a subscriber number, which should be quite random under normal circumstances.

Based on the above, we believe that conflicts through Files tend to be rare. As a starting point we therefore make the following approximation: conflicts caused by simultaneous access to the same file does not occur! This is of course an underestimation of the actual number of conflicts. The usage of the common variables in each examined block is shown in Table C.1 (CHVIEW), Table C.4 (LAD), Table C.6 (MFM), and Table C.8 (MSCCO). The conflict matrixes derived from the above tables are found in Table C.11 (CHVIEW), Table C.16 (LAD), Table C.19 (MFM), and Table C.22 (MSCCO). Applying the 'w-optimization' described in the previous section on the above conflict matrixes result in Table C.12 (CHVIEW), and Table C.17 (LAD). For the matrixes in Table C.19 and Table C.22 no improvement is achieved. This is due to that several signals share not only one, but several variables that are used for communication, e.g., ”the current state of the system is X”.

The results so far is summarized in Table 5.3.

On the other hand, by regarding Files from the other extreme, i.e., by considering **every** simultaneous access as a potential conflict, we achieve a *safe upper bound* on the number of conflicts. The usage of files in each block is shown in Table C.2 (CHVIEW), Table C.5 (LAD), Table C.7 (MFM), and Table C.9 (MSCCO). The corresponding conflict matrixes is shown Table C.13 (CHVIEW), Table C.18 (LAD), Table C.20 (MFM), and Table C.23 (MSCCO). The safe upper bound is achieved by combining Table C.12 and Table C.13 into Table C.14 (CHVIEW), Table C.17 and Table C.18 into Table C.18[1] (LAD), Table C.19 and Table C.20 into Table C.21 (MFM), and Table C.22 and Table C.23 into Table C.24 (MSCCO). Adding the upper bounds to Table 5.3 gives us Table 5.4.

Coming this far, the question is whether or not we can tighten the derived

[1] The combination of Table C.17 and Table C.18 is identical with Table C.18 since the remaining conflicts in the former are already captured in the latter.

Block	**Initial approx.**	$\backslash w$
CHVIEW	10.78%	10.74%
LAD	47.22%	8.33%
MFM	64.67%	64.67%
MSCCO	55.46%	55.46%

Table 5.3: *A first summary of the (relative) number of possible conflicts, between the observed signals. Column 2 shows our initial approximation where we assume that conflicts due to simultaneous access to the same file does not occur. In Column 3 the 'w-optimization' (working on a local copy of a variable) is applied.*

Block	**Initial approx.**	$\backslash w$	**Upper bound**
CHVIEW	10.78%	10.74%	72.56%
LAD	47.22%	8.33%	33.33%
MFM	64.67%	64.67%	75.78%
MSCCO	55.46%	55.46%	90.96%

Table 5.4: *Adding the upper bound of the possible number of conflicts (column 4) to the figures from Table 5.3.*

upper bounds. Earlier in this section we said that records in a file are used to hold subscriber data and/or data generated during a telephone call. But to prevent arbitrary accesses to a record, an instance variable (the 'state') indicates whether or not the record is currently used.

- To `SEIZE` a record is the operation of changing the state of a record from `IDLE` to `BUSY`, where `IDLE` means "not currently used by any job", and `BUSY` "currently used to hold data".

Further on, we also know (from [10]) that files (and their records) can be divided into different "sub-classes". But before we look into these sub-classes, we need to cover the concept of "Forlopps" introduced in [10].

In Chapter 2.2, we introduced the notions of jobs, and job-trees (the set of jobs originating from the same external signal);

- *a Forlopp is the set of one, or more, related job-trees* [10].

i.e., a set of job-trees that co-operate to establish, and carry out, a telephone call. Fig. 5.2 - 5.4 (all[2] from [10]) illustrate the concepts.

Having covered the Forlopp-concept, we return to the discussion on files/records[3], and start dividing the files into the following classes:

Forlopp unique: *a Forlopp unique record is a communication channel with its liveness limited to the boundary of the currently executed Forlopp. The communication channel is not live when entering or exiting the Forlopp, and the pointer value addressing the record is solely used by the Forlopp that has seized the record*, Definition 10.2 in [10]. Since the jobs in the job-trees, as well as the job-trees in the Forlopp, are sequentially executed, the only kind of conflict that can possibly occur in a Forlopp unique record, is if two different Forlopps simultaneously `SEIZE` a new record for future use. This implies that if we can classify a file as Forlopp unique, we can be sure that conflicts in the corresponding records can never occur **as long as** the SEIZE operation is protected!

Forlopp shared: similar to the Forlopp unique file, records in a "Forlopp shared" file is only used inside a Forlopp. But unlike the previous case the records are used for communication between different Forlopps which means that conflicts might occur in these records even if the SEIZE operation is secured.

[2] Fig. 5.2 is originally from [18].

[3] In the following, we will sometimes use the terms files, and records interchangeably

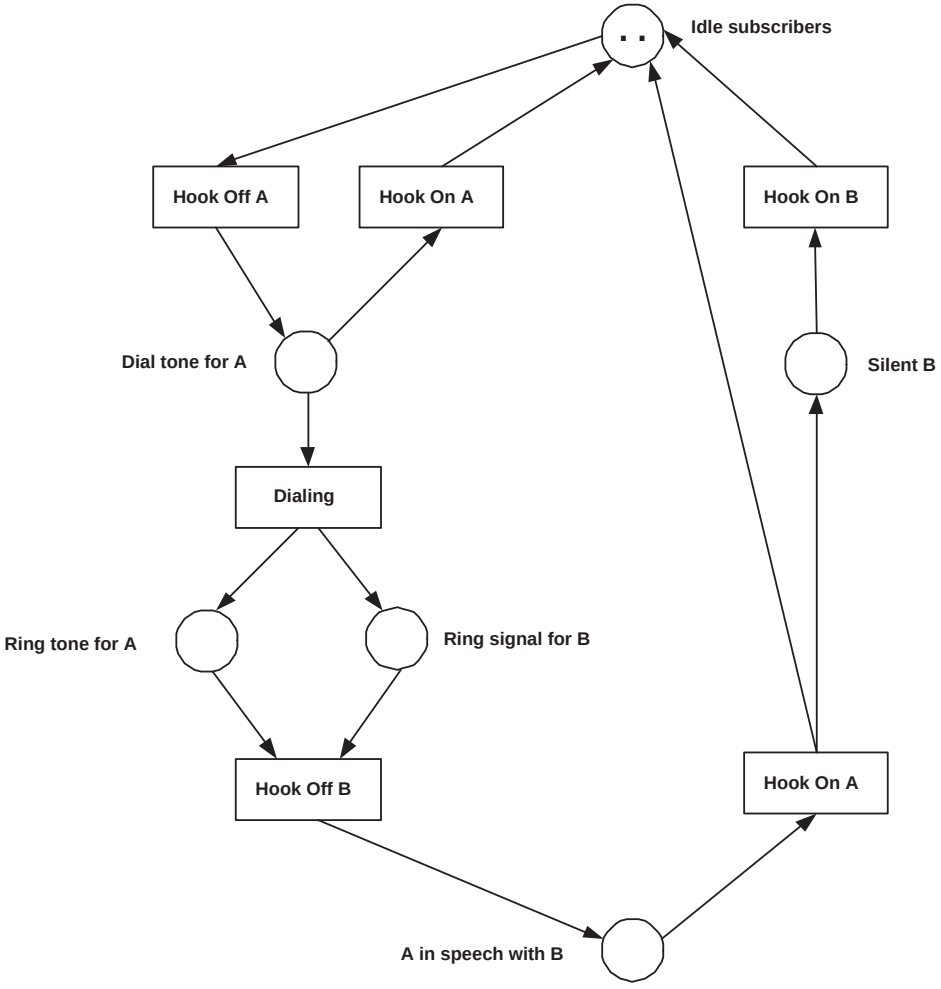

Figure 5.2: *A Petri-Net inspired representation of a Forlopp.*

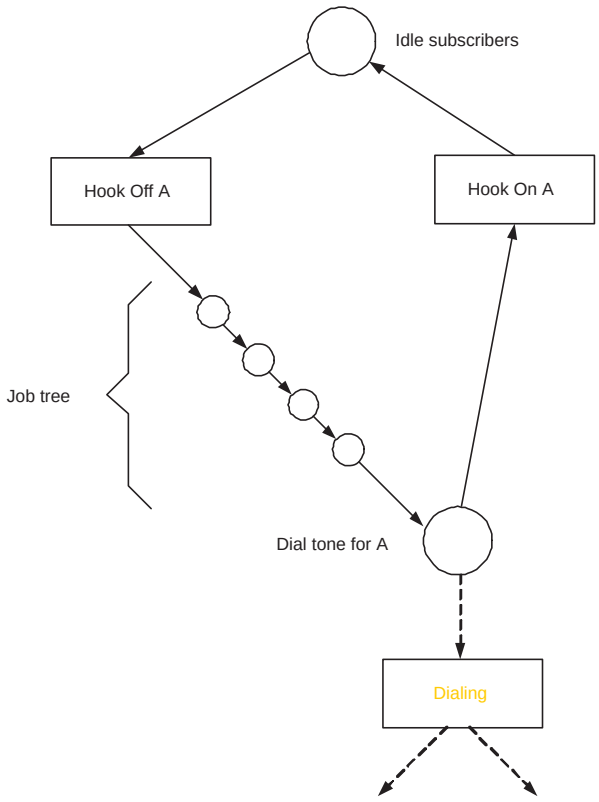

Figure 5.3: *Showing one of the Job-trees in the Forlopp from Fig. 5.2.*

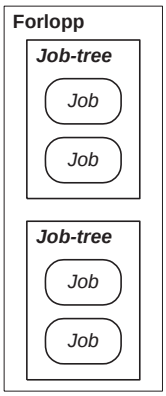

Figure 5.4: *The relation between jobs, job-trees, and forlopps.*

Shared: a record used for communication between an arbitrary number of jobs not part of a Forlopp. It might also be the case that uses of the record rely on sequential execution, e.g., by using a constant value as a pointer value. Finally, whenever we safely can't determine any of the other two cases (Forlopp unique/shared), the record must be regarded as shared. Simultaneous access to this file type must always be treated as a potential conflict!

Examining the usage of the files leads us to Table C.10 where each file is classified according to the above division. Notable is that we actually can classify 18 of 21 files as Forlopp unique, which means that **if** different signals are prevented from executing the `SEIZE` operation concurrently, there would be no conflicts in the 18 files! The `SEIZE` operation itself is a quite small piece of code, usually written in one of the following two cases

$$pointer = Commonvar;\ Commonvar = pointer : next$$

`FOR FIRST` *pointer* `FROM` *expression* `WHERE STATE = IDLE DO STATE = BUSY`

Moreover, the `SEIZE` operation is not seldom placed in a sub-routine since many signals may perform the same operation. This means that protecting the `SEIZE` operation is a question of ensuring exclusive access to the sub-routines in question! An idea that has been discussed with our partners at Ericsson is to

use the (for PLEX programmers) well known DISABLE/ENABLE construct in the following way:

```
DISABLE PARALLEL
 seize operation
ENABLE PARALLEL
```

The DISABLE/ENABLE construct is today used by code of lower priority to prevent interference from higher prioritized code[4] like in the following code snippet (and we refer to [9] for further information on the DISABLE/ENABLE construct).

```
DISABLE INTERRUPT
 low prioritized code accessing shared data
ENABLE INTERRUPT
```

Now, if we assume that the SEIZE operations in each block is protected, not only can we safely say that potential conflicts in 18 of the 21 files are eliminated, but for 2 of the examined 4 blocks we also manage to remove **all** the remaining conflicts in the common variables! This is due to the fact that these variables are (1) only used in SEIZE operations, or (2) only accessed in the sub-routines that performs the SEIZE operations!

Based on the above assumption and discussion, as well as on our earlier examination of the common variables, we conclude our study by noting that in block CHVIEW, we only need to consider the potential conflicts in Table C.3. All potential conflicts in the remaining files, as well as in the common variables, are removed under the above assumption. For block LAD we achieve even better results; **all** remaining conflicts are removed since all files are Forlopp unique, and the remaining common variables are used in SEIZE operations or will be protected if the SEIZE operations are protected. For block MFM, no file conflicts can be removed since all files are potentially shared. This means that the previous derived upper bound is our final result for the block. In block MSCCO, all considered files are classified as Forlopp unique which means, opposite to MFM, that our initial approximation is the final result for this block (since no conflicts in the common variables can be removed).

[4] We recall from Chapter 2.2 that different jobs execute on different levels of priority, and that jobs of higher priority normally are allowed to interrupt a job of lower priority.

Our final figures on the possible number of shared-memory conflicts are as in Table 5.5, and the new conflict matrix for block CHVIEW is shown in Table C.15.

Block	**Initial approx.**	$\backslash w$	**Upper bound**	**Protected SEIZE**
CHVIEW	10.78%	10.74%	72.56%	16.30%
LAD	47.22%	8.33%	33.33%	0%
MFM	64.67%	64.67%	75.78%	75.78%
MSCCO	55.46%	55.46%	90.96%	55.46%

Table 5.5: *Our final results of the potential shared-memory conflicts in the examined blocks. Our initial approximation, with and without the 'w-optimization' applied is visible in column 2, and 3 respectively. The upper bound that was later derived is shown in column 4. Column 5 captures the result when we assume a protected* `SEIZE` *operation, i.e., an atomic section solution. Notable is that the results for the LAD block is better in the last column than in our initial approximation. The reason is that the variables causing the remaining conflicts in column 3 are used in the* `SEIZE` *operations that we have assumed are sequentially executed.*

Chapter 6

Related Work

Since this thesis basically consists of two main parts; an operational semantics for Core PLEX, Chapter 4, and a case study of potential shared memory conflicts, Chapter 5, we have chosen to divide this chapter into two sections. Section 6.1 relates to the material in Chapter 4, whereas Section 6.2 relates to Chapter 5.

6.1 Semantics

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 specialized programming and specification languages are used, which have been formalized with different techniques.

CHILL (the CCITT High Level Language) is an object-oriented language with support for concurrency [19, 20]. It was developed within a denotational framework called the Vienna Development Method (VDM) [21, 22], 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 [23], has been specified by a structural operational semantics as part of a larger framework for formal reasoning about ERLANG programs [24]. ERLANG is parallel by design, and an experimental, multithreaded ERLANG implementation exists on which ERLANG programs can be directly executed without any modification [25]. A distributed version, in which message reordering and disconnecting nodes can

be expressed, is presented in [26].

Estelle, LOTOS, and SDL are specification languages proposed by, and used in, the telecom industry [27]. 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 [28], whereas the semantics for both Estelle, and LOTOS, is modeled by transition systems where the meaning is given by their computations [29, 30].

PLEX is an event-based asynchronous language. There are several event-based languages with a synchronous communication paradigm, like for instance SIGNAL [31]. However, their synchronous nature make them quite 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 harder to define a structural operational semantics for PLEX, and compositional reasoning becomes harder. However, Saabas and Uustalu [32] have recently presented a compositional, natural semantics for a language with jumps. This kind of semantics could probably be used for PLEX as well.

6.2 Concurrency Control

Due to its event-based execution model, it may seem natural to relate the possibility of parallel execution of existing PLEX programs to other event-based systems, and especially to Rational Rose RT-models since PLEX and Rose have a similar asynchronous communication paradigm with events encoded as signals [33]. However, the few papers that we are aware of in the event-based domain, [34, 35] and [36], are all concerned with optimizing performance on a single-processor architecture. Since different modeling languages such as UML and Rose are basically used in the OO domain, the lack of literature might be caused by known difficulties to parallelize OO programs (inheritance, late binding, encapsulation and reusability) [37].

As seen in previous chapters, 1 and 2.2, we have related the execution of a job to the execution of a transaction, and we will therefore review relevant works in the field of parallel databases. First of all we note that there are two architectural "extremes"; the *shared-nothing* (SN) and the *shared-memory* (SM) architectures [38, 39, 40]. The only way two processors communicate in the SN-architecture is by message passing, and hence transactions can not

interfere with each other. The SM-architectures resolve the problem with interfering transactions by locking schemes [41]. Due to better scaling and non-interference between transactions the SN-architecture has been considered superior to the SM-architecture. However, the emerge of multi core architectures will most likely force the database community to revisit the SM-architecture [42]. The latter work explores a parallel database implemented on a Cray MTA-2. This architecture provides hardware primitives for locking of single words of memory, and hashes the physical address space to distribute memory references.

The current approach to keep the system consistent is the coarse locking scheme (lock an entire block) which was described in Chapter 3. The static analysis described in the previous section is able to safely state that some of the potential conflicts never occur, which implies that the current locking scheme is unnecessarily conservative. However, potential conflicts that we can't resolve still need to be handled dynamically. An alternative approach might be non-blocking synchronization, which could be either lock-free or wait-free. The difference between these approaches is that in the lock-free approach, at least one operation is guaranteed to finish, whereas in the wait-free approach, each operation is guaranteed to finish within some time t. The wait-free approach will however impose a larger overhead due to helping scheme involved. The synchronization operation is in both cases implemented through standard hardware primitives. Examples of implementations, and algorithms, for non-blocking synchronization can be found in [43, 44, 45].

Another solution that may seem attractive (especially since we, in Chapter 1, related PLEX jobs to transactions) is Transactional Memory (TM). In this approach, operations on the shared data are seen as transactions that are either committed or aborted. The approach was originally presented as a hardware approach [46], but due to the lack of support for TM in existing hardware, [47] proposed a software based approach. Although there is currently much focus on TM, we are not aware of any existing, commercial, hardware that supports TM. Nor are we aware of any efficient (with a minimum of overhead) software based implementations.

Chapter 7

Conclusions

With the emerge of multi-core computers, existing sequential software face a major challenge. If not parallelized, the applications will probably face decreased performance since the individual cores become simpler (with lower clock frequency).

In this thesis, we have studied the event-based language PLEX, used to program the AXE telephone exchange system from Ericsson. Currently, there are approximately 20 Mlines of PLEX code in the AXE system. The software consists of independent activities (jobs), which communicate through shared data areas. Due to the large amount of code, manual rewriting of the entire code base is out of question. Thus, there is a need for automatic (or semi-automatic) methods to migrate the code to parallel architectures with a minimum of rewriting.

Our approach to migrating PLEX to a parallel architecture is based on quite standard program analysis. The goal of this analysis is to classify parallel execution as safe (or unsafe). For code that is classified as unsafe, program transformations are planned. However, due to the high availability demands that exists for telecom systems, the analysis (as well as the transformations) need to be based on a formal semantics for PLEX. This is because we need to ensure that the proposed analysis and transformations are safe.

This thesis takes the first step towards migrating PLEX to a parallel architecture, in that it provides the necessary formal basis for the analysis and the transformations by specifying an operational semantics for PLEX. We model

the execution of PLEX in the current, single-processor, architecture, as well as on a parallel implementation. The (hypothetical) parallel implementation consist of a shared-memory architecture equipped with a run-time system that executes PLEX as it is. In both cases, the operational semantics is given in terms of state transitions.

The semantics shows a straight-forward operational semantics for an imperative, non-toy like, language, with asynchronous communication through queues. The modeling of PLEX also shows an application of formal semantics that has considerable practical interest.

A second step is the case study of the potential memory conflicts that may arise when the existing code is allowed to be executed in parallel. Initially, we had to assume the worst case scenario; i.e., a conflict rate close to 100% between the examined signals. However, we have shown that simple static methods are sufficient to resolve many of the potential conflicts. As shown, we managed to approximate the figures to be in the range 11-65%. We then derived a safe upper bound on the number of potential conflicts. These figures were found to be in the range 33-91%. Under the assumption that some identified, critical, operations are prevented from parallel execution, we showed that it is possible to tighten the upper bounds to figures in the range 0-76%.

7.1 Future Work

The continuation of our work includes a formal specification, as well as an implementation, of the program analysis discussed in Chapter 1.2. As a basis for the analysis, we will take our manual inspection of the code from Chapter 5. For the proposed transformations/optimizations in Chapter 5.1, we have so far used an informal reasoning for correctness. A more formal reasoning for these, as well as for other transformations, is also on the agenda.

To maintain consistency of the shared data in the case the static analysis fails to resolve a conflict, a dynamic solution is required. We have so far compared our static analysis with a dynamic approach, where each shared data area is protected by a lock. However, we have seen that such a “mutual exclusion” approach is too conservative since two jobs accessing the same block may never touch the same data. As an alternative to this coarse grained locking scheme, we have sketched on an ’Atomic Section’ solution. This may be viewed as a pessimistic, but more fine grained approach, than the current. Another approach is non-blocking synchronization as discussed in Chapter 6.2.

This will probably involve a re-design of some existing data structures in order to make them suitable for concurrent access.

Regardless of the approach(es) chosen, evaluation with respect to expected increased performance is planned as the final step of the continuation of our work.

Finally, in Chapter 6.2 we discarded Transactional Memory as a solution (even though there is a clear relation between PLEX jobs and transactions) due to the lack of support for TM in existing hardware. A software TM solution was also rejected since we are not aware of any efficient algorithms. However, if the situation would change in a near future, we might reconsider TM as a dynamic solution.

Bibliography

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

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

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

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

[5] Jesse Z. Fang. Parallel programming environment: a key to translating tera-scale platforms into a big success. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–1, New York, NY, USA, 2007. ACM.

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

[7] Andres S Tanenbaum. *Distributed Operating Systems*. Prentice Hall International Editions, 1995.

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

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

[10] B. Lindell. Analysis of reentrancy and problems of data interference in the parallel execution of a multi processor AXE-APZ system. Master's thesis, Mälardalen University, 2003.

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

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

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

[15] A. Arnstrom, 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.

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

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

[18] Peter Funk and Janet Wennersten. Asynchronous signal paradigm and AI for soft real time systems. Technical report, Mälardalen University, March 2000.

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

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

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

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

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

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

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

[26] Hans Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed erlang. In *Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] Rational. *Modeling Language Guide - Rational Rose Realtime*, 2002.

[34] A. Marburger and D. Herzberg. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 139 – 147, 2001.

[35] Christof Mosler. E-CARES Project: Reengineering of PLEX Systems. *Softwaretechnik-Trends*, 26(2):59–60, 5 2006.

[36] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation PLDI '02*, pages 106 – 116, 2002.

[37] S. Kumar. Issues in parallelizing object-oriented programs. In *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pages 64–71, 1995.

[38] David J. DeWitt and Jim Gray. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.*, 19(4):104–112, 1990.

[39] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, 1996.

[40] Ameet S. Talwadker. Survey of performance issues in parallel database systems. *J. Comput. Small Coll.*, 18(6):5–9, 2003.

[41] Paul Watson and George Catlow. Architecture of the ICL goldrush megaserver. In *BNCOD 13: Proceedings of the 13th British National Conference on Databases*, pages 249–262, London, UK, 1995. Springer-Verlag.

[42] John Cieslewicz, Jonathan Berry, Bruce Hendrickson, and Kenneth A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *DaMoN '06: Proceedings of the 2nd international workshop on Data management on new hardware*, page 4, New York, NY, USA, 2006. ACM Press.

[43] Yi Zhang. *Non-Blocking Synchronization: Algorithms and Performance Evaluation*. PhD thesis, Chalmers University of Technology, 2003.

[44] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.

[45] Phuong Ha. *Reactive Concurrent Data Structures and Algorithms for Synchronization*. PhD thesis, Chalmers University of Technology, 2006.

[46] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[47] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

Appendix A

The Sequential Semantics for Core PLEX

This chapter summarizes the sequential semantics for Core PLEX that was given in Chapter 4.3.

[ass] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{x:=a} \langle succ(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma],\ \delta\rangle$

[cond^{tt}] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2} \langle succT(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma,\ \delta\rangle$ **if** $\mathcal{B}[\![b]\!]\sigma = \texttt{tt}$

[cond^{ff}] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2} \langle succF(\mathcal{VSC}),\ JBA,\ JBB,\ \sigma,\ \delta\rangle$ **if** $\mathcal{B}[\![b]\!]\sigma = \texttt{ff}$

[jmp] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{GOTO}\ label} \langle label,\ JBA,\ JBB,\ \sigma,\ \delta\rangle$

[send^{dir}] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto \bot,\ \delta\rangle$ if $signal \in \mathbf{Dir}$

$$[\text{send}^{dir}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta\rangle \quad \text{if} \quad signal \in \mathbf{Dir}$$

$$[\text{send}^{buf}_{LevA}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal} \langle succ(\mathcal{VSC}),\ JBA : (signal, \perp),\ JBB,\ \sigma,\ \delta\rangle \quad \text{if} \quad signal \in \mathbf{Buf},\ signal \in \mathbf{LevA}$$

$$[\text{send}^{buf}_{LevA}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data} \langle succ(\mathcal{VSC}),\ JBA : (signal, data),\ JBB,\ \sigma,\ \delta\rangle \quad \text{if} \quad signal \in \mathbf{Buf},\ signal \in \mathbf{LevA}$$

$$[\text{send}^{buf}_{LevB}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal} \langle succ(\mathcal{VSC}),\ JBA,\ JBB : (signal, \perp),\ \sigma,\ \delta\rangle \quad \text{if} \quad signal \in \mathbf{Buf},\ signal \in \mathbf{LevB}$$

$$[\text{send}^{buf}_{LevB}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data} \langle succ(\mathcal{VSC}),\ JBA,\ JBB : (signal, data),\ \sigma,\ \delta\rangle \quad \text{if} \quad signal \in \mathbf{Buf},\ signal \in \mathbf{LevB}$$

$$[\text{send}^{comb}_{fw}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label} \langle cfsig,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto \perp,\ label : \delta\rangle$$

$$[\text{send}^{comb}_{fw}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label} \langle cfsig,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data,\ label : \delta\rangle$$

$$[\text{send}^{comb}_{bw}] \quad \langle \mathcal{VSC},\ JBA,\ ,JBB,\ \sigma,\ label : \delta\rangle \xrightarrow{\texttt{RETURN}\ cbsig} \langle label,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto \perp,\ \delta\rangle$$

$$[\text{send}^{comb}_{bw}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ label : \delta\rangle \xrightarrow{\texttt{RETURN}\ cbsig\ \texttt{WITH}\ data} \langle label,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data,\ \delta\rangle$$

$$[\text{send}^{local}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{TRANSFER}\ signal} \langle signal,\ JBA,\ JBB,\ \sigma,\ \delta\rangle$$

$$[\text{send}^{local}] \quad \langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{TRANSFER}\ signal\ \texttt{WITH}\ data} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM_i} \mapsto data \setminus \perp,\ \delta\rangle$$

[exit] $\langle \mathcal{VSC},\ (signal, data) : JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{EXIT}} \langle signal,\ JBA,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta\rangle$

[exit] $\langle \mathcal{VSC},\ \bot,\ (signal, data) : JBB,\ \sigma,\ \delta\rangle \xrightarrow{\texttt{EXIT}} \langle signal,\ \bot,\ JBB,\ \sigma|_{RM} \mapsto data,\ \delta\rangle$

[ext] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\epsilon} \langle \mathcal{VSC},\ JBA : (signal, data),\ JBB,\ \sigma,\ \delta\rangle$
if $signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevA}$

[ext] $\langle \mathcal{VSC},\ JBA,\ JBB,\ \sigma,\ \delta\rangle \xrightarrow{\epsilon} \langle \mathcal{VSC},\ JBA,\ JBB : (signal, data),\ \sigma,\ \delta\rangle$
if $signal \in \mathbf{Ext},\ \ signal \in \mathbf{LevB}$

Appendix B

The Parallel Semantics for Core PLEX

Whereas Chapter A summarized the sequential semantics for Core PLEX (given in Chapter 4.3), this chapter summarizes the parallel semantics given in Chapter 4.4.

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

[cond^{tt}] $$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i} \langle succT(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

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

[cond^{ff}] $$\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2,\ i} \langle succF(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$$

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

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

$$[\text{send}^{dir}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i} \langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

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

$$[\text{send}^{dir}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i} \langle signal, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

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

$$[\text{send}^{buf}_{LevA}] \quad \langle s_1, \ldots, s_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i} \langle s'_1, \ldots, s'_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$
$$\begin{aligned} where \quad & s_1 = \langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1 \rangle \\ & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_1 = \langle \mathcal{VSC}_1,\ JBA : (signal, \bot),\ JBB_1,\ Locks_1, \\ & \qquad \mathcal{F}_1 : [signal],\ \delta_1 \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \end{aligned}$$

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

$$[\text{send}^{buf}_{LevA}] \quad \langle s_1, \ldots, s_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i} \langle s'_1, \ldots, s'_i, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$
$$\begin{aligned} where \quad & s_1 = \langle \mathcal{VSC}_1,\ JBA,\ JBB_1,\ Locks_1,\ \mathcal{F}_1,\ \delta_1 \rangle \\ & s_i = \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \\ & s'_1 = \langle \mathcal{VSC}_1,\ JBA : (signal, data),\ JBB_1,\ Locks_1, \\ & \qquad \mathcal{F}_1 : [signal],\ \delta_1 \rangle \\ & s'_i = \langle succ(\mathcal{VSC}_i),\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i \rangle \end{aligned}$$

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

$$[\text{send}^{buf}_{LevB}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [\mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal,\ i} \langle succ(\mathcal{VSC}_i), JBB_i : (signal, \bot),\ Locks_i, [\mathcal{T} : signal] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

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

$$[\text{send}^{buf}_{LevB}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{WITH}\ data,\ i} \langle succ(\mathcal{VSC}_i), JBB_i : (signal, data),\ Locks_i, [\mathcal{T} : signal] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

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

$$[\text{send}^{comb}_{fw}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label,\ i} \langle cfsig, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ label : \delta_i, \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

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

$$[\text{send}^{comb}_{fw}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ cfsig\ \texttt{WITH}\ data\ \texttt{WAIT FOR}\ cbsig\ \texttt{IN}\ label,\ i} \langle cfsig, JBB_i, Locks_i \cup \{L_\gamma\},\ \mathcal{F}_i,\ label : \delta_i, \langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$$

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

$$[\text{send}^{comb}_{bw}] \quad \langle \mathcal{VSC}_i, JBB_i, Locks_i,\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{RETURN}\ cbsig,\ i} \langle label, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto \bot, \sigma_L \rangle \rangle$$

$$[\text{send}^{comb}_{bw}] \quad \langle \mathcal{VSC}_i, JBB_i, Locks_i,\ \mathcal{F}_i,\ label : \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{RETURN}\ cbsig\ \texttt{WITH}\ data,\ i} \langle label, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma|_{RM_i} \mapsto data, \sigma_L \rangle \rangle$$

$$[\text{send}^{local}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{TRANSFER}\ signal,\ i} \langle signal, JBB_i, Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

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

$$[\text{send}^{buf}_{jtree}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE},\ i} \langle succ(\mathcal{VSC}_i), JBB_i : (signal, \bot),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i, \langle \sigma, \sigma_L \rangle \rangle$$

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

$$[\text{send}^{buf}_{jtree}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE WITH}\ data,\ i} \langle succ(\mathcal{VSC}_i), JBB_i : (signal, data),\ Locks_i,\ \mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma, \sigma_L \rangle \rangle$$

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

$$[\text{send}^{buf}_{jtree}] \quad \langle \ldots, s_i, s_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE},\ i} \langle \ldots, s'_i, s'_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

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

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

$$[\text{send}^{buf}_{jtree}] \quad \langle \ldots, s_i, s_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle \xrightarrow{\texttt{SEND}\ signal\ \texttt{JOBTREE WITH}\ data,\ i} \langle \ldots, s'_i, s'_j, \ldots, \langle \sigma, \sigma_L \rangle \rangle$$

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

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

$$[\text{exit}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [\,] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{EXIT},\ i} \langle \bot, JBB_i, \emptyset,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle \rangle$$

$$[\text{exit}] \quad \langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\texttt{EXIT},\ i} \langle \bot, JBB_i, \emptyset,\ [signal : \mathcal{T}] : \mathcal{F}_i,\ \delta_i, \langle \sigma, \sigma_L[L_\gamma \mapsto 0, L_\gamma \in Locks_i] \rangle \rangle$$

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

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

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

if $Receiver(signal) = j \neq i$

[job] $\langle \bot,\ (signal, data) : JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$
$\langle signal, JBB_i, \{L_\gamma\},\ [\mathcal{T}] : \mathcal{F}_i\ -\ [signal : \mathcal{T}],\ \delta_i,$
$\langle \sigma|_{RM_i} \mapsto data, \sigma_L[L_\gamma \mapsto 1] \rangle \rangle$

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

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

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

[ext$_{LevB}$] $\langle \mathcal{VSC}_i,\ JBB_i,\ Locks_i,\ \mathcal{F}_i,\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle \xrightarrow{\epsilon, i}$
$\langle \mathcal{VSC}_i,\ JBB_i : (signal, data),\ Locks_i,$
$\mathcal{F}_i : [signal],\ \delta_i,\ \langle \sigma,\ \sigma_L \rangle \rangle$

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

[global] $\dfrac{s_i \longrightarrow s'_i}{\langle s_1,\ \ldots\ s_i,\ \ldots\ ,\ s_k,\ s_G \rangle \rightarrow \langle s_1,\ \ldots\ s'_i,\ \ldots\ ,\ s_k,\ s_G \rangle}$

[global] $\dfrac{s_i \longrightarrow s'_i}{\langle s_1,\ \ldots\ s_i,\ \ldots\ ,\ s_k,\ s_G \rangle \rightarrow \langle s_1,\ \ldots\ s'_i,\ \ldots\ ,\ s_k,\ s'_G \rangle}$

Appendix C

The Potential Memory Conflicts

The background material, which we refer to in Section 5.2, and from which the figures in Table 5.3-5.5 is derived, is collected in this section. Table C.1-C.9 shows how the variables in each examined block are used, and from Section 5.1, we repeat our classification of the variables;

- $\perp$ - The variable is **never** used by the signal in question.
- R - Read Only, i.e., the only way the signal is accessing the variable is in read operations.
- W - If the signal accesses the variable, the first access will **always** be a write operation.
- $\top$ - It is not possible to (statically) classify the variable according to the previous cases.

Table C.10 contains the classification of the files, whereas Table C.11-C.24 contains the conflict matrixes for the different blocks.

When the figures in Table 5.3-5.5 was calculated, it was only necessary to consider one half in each respective conflict matrix. This is (of course) due to symmetry; if $Signal_A$ may be in conflict with $Signal_B$, then obviously $Signal_B$ may be in conflict with $Signal_A$. However, $Signal_A$ may also, in some situations, be in conflict with itself. This has been included in our figures.

Finally, some of the signals in the tables are marked bold, and italic; these are the ”high-frequent” signals mentioned in the beginning of Section 5.2.

Block: CHVIEW (Middleware) *variables →* *signals ↓*	CPUBDATAID	CPUBDATAVALUE	CPUB2DATAID	CPUB2DATAVALUE	CGLOBALSUBIUA	CGLOBALSUBSREF	CSTARTIDLELISTLOGPRT	CSTARTIDLELISTVIEW	CSTARTIDLELISTLOGHDR	CSETMASK	CGLOBALSUBPCDI	CTMZSUPPORTED	CAUTOSIZEALTSUPPORT	COWNREF	CSTOPIDLELISTLOGPRT	CPREPNUMLOGPART	CSTOPIDLELISTVIEW	CSTOPIDLELISTLOGHDR	CPUBLICDEFAULTS	CCLEARSUBSCR
WRITEPUBLIC	W	W	⊤	⊤	R	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
READPUBLIC	W	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
READLOG	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
CREGENVIEW	⊥	⊥	⊥	⊥	⊥	R	⊤	⊤	⊤	R	R	R	R	R	W	⊥	W	W	R	R
WRITELOGEND	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
CREATEVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊤	R	⊥	R	R	R	W	⊥	W	W	⊥	⊥
INITVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊤	W	W	R	⊥	⊥	R	R	⊤	R	⊤	⊤	⊥	⊥
OPENVIEWCON	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊤	R	⊥	⊥	R	R	W	⊥	⊥	W	⊥	⊥
WRITEPUBEND	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
WRITELOG	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
RESERVEVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊥	⊥	R	R	R	⊥	⊥	W	W	⊥	⊥
STOREFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
OPENSESSION1R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OPENSESSION1REJ	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	R	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
INITOUTPUT	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	R	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
CONTINUEB	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
REQRESKEY	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
STARTCHOUTPUT	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	R	R	⊥	⊥	⊥	⊥	⊥	⊥
PARTOUTBREAK	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
FOAMTRACE	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RELVIEWCON	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
CANLOCPUBLIC1	⊥	⊥	⊥	⊥	⊥	⊥	W	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥
UNDOWRITELOG	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊥	⊥	⊥	⊥
CLEARLOG	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊥	⊥	⊥	⊥
PARTOUTREQ	⊥	⊥	⊥	⊥	⊥	⊥	W	⊤	⊤	⊥	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
OUTPUTVAREQR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊥	⊥	⊥	⊥	R	⊥	⊥	W	W	⊥	⊥
DISASSOCVIEW	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
DELAYDISASSOC	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
SPLITVA	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
DELAYSPLITVA	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
NOOUTPUTR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
CHARGRESULTSR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
PARTOUTREADYR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
SETPRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
PRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLEARPRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
PRECONGFEPCLRD	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTVAREQREJ	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTFEPR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTFEPREJ	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLOSESESSION1R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLEARFULLCONG	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
GETSTRUCTOT	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SWITCHFEPS	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
FEPSWITCHEDR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥

Table C.1: CHVIEW - usage of common variables.

Block: CHVIEW (Middleware) *Files* → *signals* ↓	VIEWFILE	LOGHEADER	LOGPART	PUBLICDATA	SESSION	SAECNTRECORD	*Files* → *signals* ↓	VIEWFILE	LOGHEADER	LOGPART	PUBLICDATA	SESSION	SAECNTRECORD
WRITEPUBLIC	R	⊥	⊥	W	⊥	⊥	OUTPUTFEPR	⊤	⊤	⊤	⊥	⊤	⊤
READPUBLIC	R	⊥	⊥	R	⊥	⊥	OUTPUTFEPREJ	⊤	⊤	⊤	⊥	⊤	⊤
READLOG	⊥	R	R	⊥	W	⊥	CLOSESESSION1R	⊤	⊤	⊤	⊥	⊤	⊤
CREGENVIEW	⊤	⊤	⊤	W	⊥	⊤	CLEARFULLCONG	⊤	⊤	⊤	⊥	⊤	⊤
WRITELOGEND	⊤	W	⊤	⊥	⊥	⊤	GETSTRUCTOT	⊤	R	R	⊥	W	⊥
CREATEVIEW	⊤	⊤	⊤	⊥	⊥	⊤	CHVIEWCONTRACE	⊤	⊤	⊥	⊥	⊥	⊥
INITVIEW	W	W	⊤	⊥	⊥	⊤	CHVIEWCONCLEAR	⊤	W	⊥	⊥	⊥	⊥
OPENVIEWCON	⊤	⊤	⊤	⊥	⊥	⊤	RVIEWSTRUC	W	⊥	⊥	⊥	⊥	⊥
WRITEPUBEND	R	⊥	⊤	⊤	⊥	⊤	STORERESKEY	W	⊥	⊥	⊥	⊥	⊥
WRITELOG	⊤	W	⊤	⊥	⊥	⊤	SUBLOCPUBCDT	R	⊥	⊥	⊤	⊥	⊥
RESERVEVIEW	⊤	⊤	⊥	⊥	⊥	⊤	VIEWSTATUS	W	⊥	⊥	⊥	⊥	⊥
STOREFEP	⊤	W	⊥	⊥	⊥	⊥	VIEWSERVICE	W	⊥	⊥	⊥	⊥	⊥
OPENSESSION1R	⊤	⊤	⊤	⊥	⊤	⊤	VIEWSPEC	W	⊥	⊥	⊥	⊥	⊥
OPENSESSION1REJ	⊤	⊤	⊤	⊥	⊤	⊤	UPDATEEVENT	W	⊥	⊥	⊥	⊥	⊥
INITOUTPUT	⊤	⊤	⊤	⊥	⊤	⊤	READVABSTIME	W	⊥	⊥	⊥	⊥	⊥
CONTINUEB	⊤	⊤	⊤	⊥	⊤	⊤	ASSOCVIEW	⊤	R	⊥	⊥	⊥	⊥
STARTCHOUTPUT	⊤	⊤	⊥	⊥	⊥	R	DELAYASSOC1	⊤	R	⊥	⊥	⊥	⊥
PARTOUTBREAK	⊤	⊤	⊥	⊥	⊥	⊥	SETVAVIEWMIS	⊤	⊥	⊥	⊥	⊥	⊥
FOAMTRACE	⊤	⊥	⊥	⊥	⊤	⊥	FEPSTREATED	⊤	⊥	⊥	⊥	W	⊥
RELVIEWCON	⊤	⊤	⊤	⊤	⊥	⊤	GETNEXTVIEW	⊤	⊥	⊥	⊥	W	⊥
CANLOCPUBLIC1	⊤	⊥	W	W	⊥	⊤	GETVABSTIME	R	⊥	⊥	⊥	W	⊥
UNDOWRITELOG	⊤	W	⊤	⊥	⊥	⊤	GETFEPPONUM	R	⊥	⊥	⊥	W	⊥
CLEARLOG	⊤	W	⊤	⊥	⊥	⊤	GETVIEWINFO	R	R	R	⊥	W	⊥
PARTOUTREQ	W	⊤	⊤	⊥	⊥	⊤	TRACEAM	R	⊥	⊥	⊥	⊥	⊥
OUTPUTVAREQR	⊤	⊤	⊥	⊥	⊥	⊤	SUBLOCPUBLIC	R	⊥	⊥	⊤	⊥	⊥
DISASSOCVIEW	⊤	⊤	⊤	⊥	⊥	⊤	CANLOCPUBLIC	R	⊥	⊥	⊤	⊥	⊥
SPLITVA	⊤	⊤	⊤	⊥	⊥	⊤	COPYPUBLIC	R	⊥	⊥	⊤	⊥	⊥
NOOUTPUTR	⊤	⊤	⊤	⊥	⊥	⊤	GETNEXTLOG	⊥	R	R	⊥	W	⊥
CHARGRESULTSR	W	⊤	⊤	⊥	⊥	⊤	STOREAC	⊥	W	⊥	⊥	⊥	⊥
PARTOUTREADYR	⊤	⊤	⊤	⊥	⊥	⊤	FOAMCLEAR	⊥	⊥	⊥	⊥	W	⊥
OUTPUTVAREQREJ	⊤	R	⊥	⊥	⊥	⊥							

Table C.2: CHVIEW - usage of File variables

Block: CHVIEW (Middleware) *Files →* *signals ↓*						SAECNTRECORD	*Files →* *signals ↓*						SAECNTRECORD
WRITEPUBLIC						⊥	OUTPUTFEPR						⊤
READPUBLIC						⊥	OUTPUTFEPREJ						⊤
READLOG						⊥	CLOSESESSION1R						⊤
CREGENVIEW						⊤	CLEARFULLCONG						⊤
WRITELOGEND						⊤	GETSTRUCTOT						⊥
CREATEVIEW						⊤	CHVIEWCONTRACE						⊥
INITVIEW						⊤	CHVIEWCONCLEAR						⊥
OPENVIEWCON						⊤	RVIEWSTRUC						⊥
WRITEPUBEND						⊤	STORERESKEY						⊥
WRITELOG						⊤	SUBLOCPUBCDT						⊥
RESERVEVIEW						⊤	VIEWSTATUS						⊥
STOREFEP						⊥	VIEWSERVICE						⊥
OPENSESSION1R						⊤	VIEWSPEC						⊥
OPENSESSION1REJ						⊤	UPDATEEVENT						⊥
INITOUTPUT						⊤	READVABSTIME						⊥
CONTINUEB						⊤	ASSOCVIEW						⊥
STARTCHOUTPUT						R	DELAYASSOC1						⊥
PARTOUTBREAK						⊥	SETVAVIEWMIS						⊥
FOAMTRACE						⊥	FEPSTREATED						⊥
RELVIEWCON						⊤	GETNEXTVIEW						⊥
CANLOCPUBLIC1						⊤	GETVABSTIME						⊥
UNDOWRITELOG						⊤	GETFEPPONUM						⊥
CLEARLOG						⊤	GETVIEWINFO						⊥
PARTOUTREQ						⊤	TRACEAM						⊥
OUTPUTVAREQR						⊤	SUBLOCPUBLIC						⊥
DISASSOCVIEW						⊤	CANLOCPUBLIC						⊥
SPLITVA						⊤	COPYPUBLIC						⊥
NOOUTPUTR						⊤	GETNEXTLOG						⊥
CHARGRESULTSR						⊤	STOREAC						⊥
PARTOUTREADYR						⊤	FOAMCLEAR						⊥
OUTPUTVAREQREJ						⊥							

Table C.3: CHVIEW - file variables that need to be considered under the assumption that SEIZE is not executed in parallel.

Block: LAD (OS) *variables* → / *signals* ↓	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
CCOMBANK512FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK512MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK512NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK512NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK512FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK256FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK256MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK256NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK256NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK256FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK128FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK128MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK128NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK128NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK128FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK64FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK64MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK64NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK64NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK64FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK32MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK16KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK8KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK4KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK2KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK1KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK512PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK256PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK128PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK64PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK32PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
COWNREF	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CAUTOSAACTIVE	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CCOMBANKRESULTCODETEMP	⊣	⊣	⊣	W	⊣	⊣	⊣	⊣
CCOMBANKUSERPOINTERTEMP	⊣	⊣	⊣	W	W	W	W	⊣
CCLOCKDATE	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CPROGERRORBUFFERTYPE	W	W	W	W	W	⊣	⊣	⊣
CPROGERRORCODE	W	W	W	W	W	⊣	⊣	⊣

Block: LAD (OS) *variables* → / *signals* ↓	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
CCOMBANKNIL	⊣	⊣	⊣	R	⊣	⊣	R	⊣
CCOMBANKTIMEOUTQUEUECARDINALITY	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣
CCOMBANKTIMEOUTQUEUEENTRY	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣
CCOMBANKTIMEOUTQUEUEEXIT	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣
CCOMBANK32KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK32KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK16KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK16KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK16KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK16KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK16KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK8KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK8KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK8KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK8KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK8KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK4KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK4KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK4KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK4KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK4KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK2KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK2KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK2KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK2KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK2KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK1KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK1KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK1KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK1KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK1KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣

Table C.4: LAD - usage of common variables.

Block: LAD (OS) *Files* → *signals* ↓	COMBANK32	COMBANK64	COMBANK128	COMBANK256	COMBANK512	COMBANK1K	COMBANK2K	COMBANK4K	COMBANK8K	COMBANK16K	COMBANK32K
ALLCOMBUF	W	W	W	W	W	W	W	W	W	W	W
CBFLCONNECT	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
GETCOMBUFREF	R	R	R	R	R	R	R	R	R	R	R
RELCOMBUF	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
READCBSTATE	R	R	R	R	R	R	R	R	R	R	R

Table C.5: LAD - usage File variables.

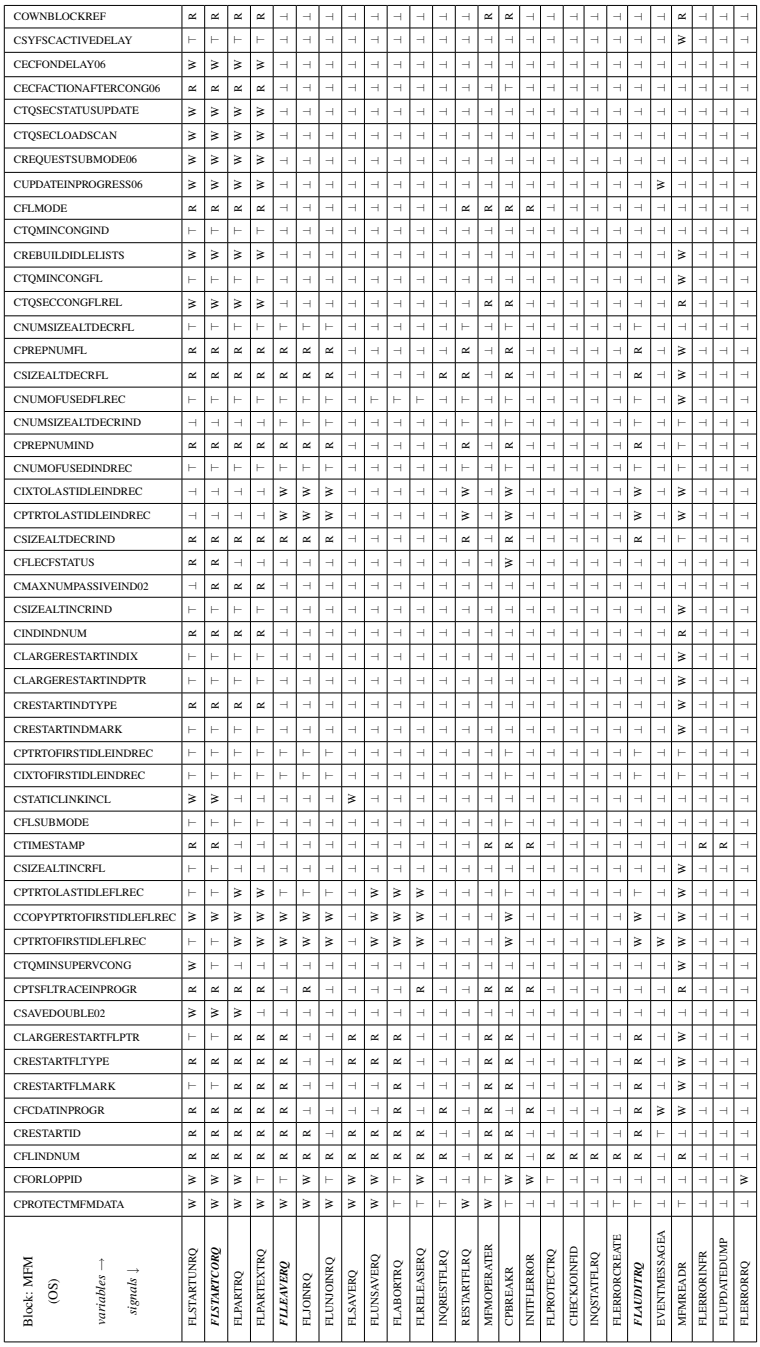

Table C.6: MFM - usage of common variables.

Block: MFM (OS) *Files* → *signals* ↓	FLREC	INDREC
FLSTARTUNRQ	⊤	⊤
FLSTARTCORQ	⊤	⊤
FLPARTRQ	⊤	⊤
FLPARTEXTRQ	⊤	⊤
FLLEAVERQ	⊤	⊤
FLJOINRQ	⊤	⊤
FLUNJOINRQ	⊤	⊤
FLSAVERQ	⊤	⊥
FLUNSAVERQ	⊤	⊥
FLABORTRQ	⊤	⊥
FLRELEASERQ	⊤	⊥
INQRESTFLRQ	R	R
RESTARTFLRQ	⊤	⊤
CPBREAKR	⊤	⊤
FLPROTECTRQ	⊤	⊥
CHECKJOINFID	R	⊥
INQSTATFLRQ	R	⊥
FLERRORCREATE	R	R
FLAUDITRQ	⊤	⊤
MFMREADR	⊤	⊤

Table C.7: MFM - usage File variables.

Block: MSCCO (Application) *variables →* *signals ↓*	CMESSAGEBUFR	CDATAREADFROMTEMP	CMESSAGEBUFRA	CANTERALENGTH	CUNEXPECTEDEXIT	CDATALENGTH	CSSN	CINDEX	CEXTSPC1	CEXTSPC2	CPRIORITYANSI	COWNSPMEM	COWNSPNET	COWNSPCLU
SCCONNIND	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
SCCONNINDE	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥	⊥
RNOCOMREQ	W	⊥	⊥	⊥	⊤	⊤	W	W	W	W	R	R	R	R
RNOCOMREQL	W	⊥	⊥	⊥	⊤	⊤	W	W	W	W	R	R	R	R
C7CREFIND2	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONF	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFIND	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2	W	W	W	⊤	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAIND	W	W	W	⊤	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFE	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2E	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDE	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2E	W	W	W	⊤	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDE	W	W	W	⊤	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2	W	⊥	W	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2E	W	⊥	W	⊥	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARET	W	⊥	W	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARETE	W	⊥	W	⊥	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CBSEIZER	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2E	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DISCIND2	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DISCIND2E	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7SCSEIZED	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7SCCONG	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDISCIND	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDISCINDE	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
BSCPTODPCR1	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥
OPCTOBSCPR	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYCONG	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYCONRESP	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYDISC	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYLINKED	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRSEIZEFAIL	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
CBSEIZEF	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥
SCCONNINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2E	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥	⊥
SCDATARETI	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCOWNSPINFO	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W
C7CONNIND2	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
SEIZECOO	R	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
C7DATARET2I	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2S	W	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARETS	W	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥

Table C.8: MSCCO - usage of common variables.

Block: MSCCO (Application) *Files* → *signals* ↓	MSCCODATADATA	TEMPRECORD	*Files* → *signals* ↓	MSCCODATADATA	TEMPRECORD
SCCONNIND	⊤	⊥	RRLAYLINKED	⊤	⊥
SCCONNINDE	⊤	⊥	RRSEIZEFAIL	⊤	⊥
RNOCOMREQ	⊤	⊥	CBSEIZEF	⊤	⊤
RNOCOMREQL	⊤	⊥	C7CONNCONF2I	⊤	⊥
C7CREFIND2	⊤	⊥	C7CREFIND2I	⊤	⊥
SCCONNCONF	⊤	⊥	SCCONNCONFI	⊤	⊥
SCCREFIND	⊤	⊥	SCCREFINDI	⊤	⊥
C7DATAIND2	⊤	⊤	C7DATAIND2I	⊤	⊥
SCDATAIND	⊤	⊤	SCDATAINDI	⊤	⊥
SCCONNCONFE	⊤	⊥	C7CONNCONF2S	R	⊥
C7CREFIND2E	⊤	⊥	SCCONNCONFS	R	⊥
SCCREFINDE	⊤	⊥	C7CREFIND2S	R	⊥
C7DATAIND2E	⊤	⊤	SCCREFINDS	R	⊥
SCDATAINDE	⊤	⊤	C7DATAIND2S	R	⊥
C7DATARET2	R	⊥	SCDATAINDS	R	⊥
C7DATARET2E	R	⊥	C7CONNIND2E	⊤	⊥
SCDATARET	⊤	⊥	SCDATARETI	R	⊥
SCDATARETE	⊤	⊥	C7CONNIND2	⊤	⊥
CBSEIZER	⊤	⊤	SEIZECOO	⊤	⊥
C7CONNCONF2	⊤	⊥	C7DATARET2I	R	⊥
C7CONNCONF2E	⊤	⊥	C7DATARET2S	R	⊥
C7DISCIND2	⊤	⊤	SCDATARETS	R	⊥
C7DISCIND2E	⊤	⊤	C7DISCIND2I	⊤	⊥
C7SCSEIZED	⊤	⊥	C7DISCIND2S	R	⊥
C7SCCONG	⊤	⊥	CBRELEASER	⊤	⊥
SCDISCIND	⊤	⊤	RSCLEAR	⊤	⊥
SCDISCINDE	⊤	⊤	RSTRACE	⊤	⊥
BSCPTODPCR1	⊤	⊥	SCDISCINDI	⊤	⊥
OPCTOBSCPR	⊤	⊥	SCDISCINDS	R	⊥
RRLAYCONG	⊤	⊥	TRACE	R	⊥
RRLAYCONRESP	⊤	⊥	FLSETSTORD	W	⊤
RRLAYDISC	⊤	⊤			

Table C.9: MSCCO - usage File variables.

Block	*File*	*Shared*	*Forlopp: shared*	*Forlopp: unique*
CHVIEW				
	VIEWFILE			X
	LOGHEADER			X
	LOGPART			X
	PUBLICDATA			X
	SESSION			X
	SAECNTRECORD	X		
LAD				
	COMBANK32			X
	COMBANK64			X
	COMBANK128			X
	COMBANK256			X
	COMBANK512			X
	COMBANK1K			X
	COMBANK2K			X
	COMBANK4K			X
	COMBANK8K			X
	COMBANK16K			X
	COMBANK32K			X
MFM				
	FLREC	X		
	INDREC	X		
MSCCO				
	MSCCODATADATA			X
	TEMPRECORD			X

Table C.10: The different types of files in the examined blocks.

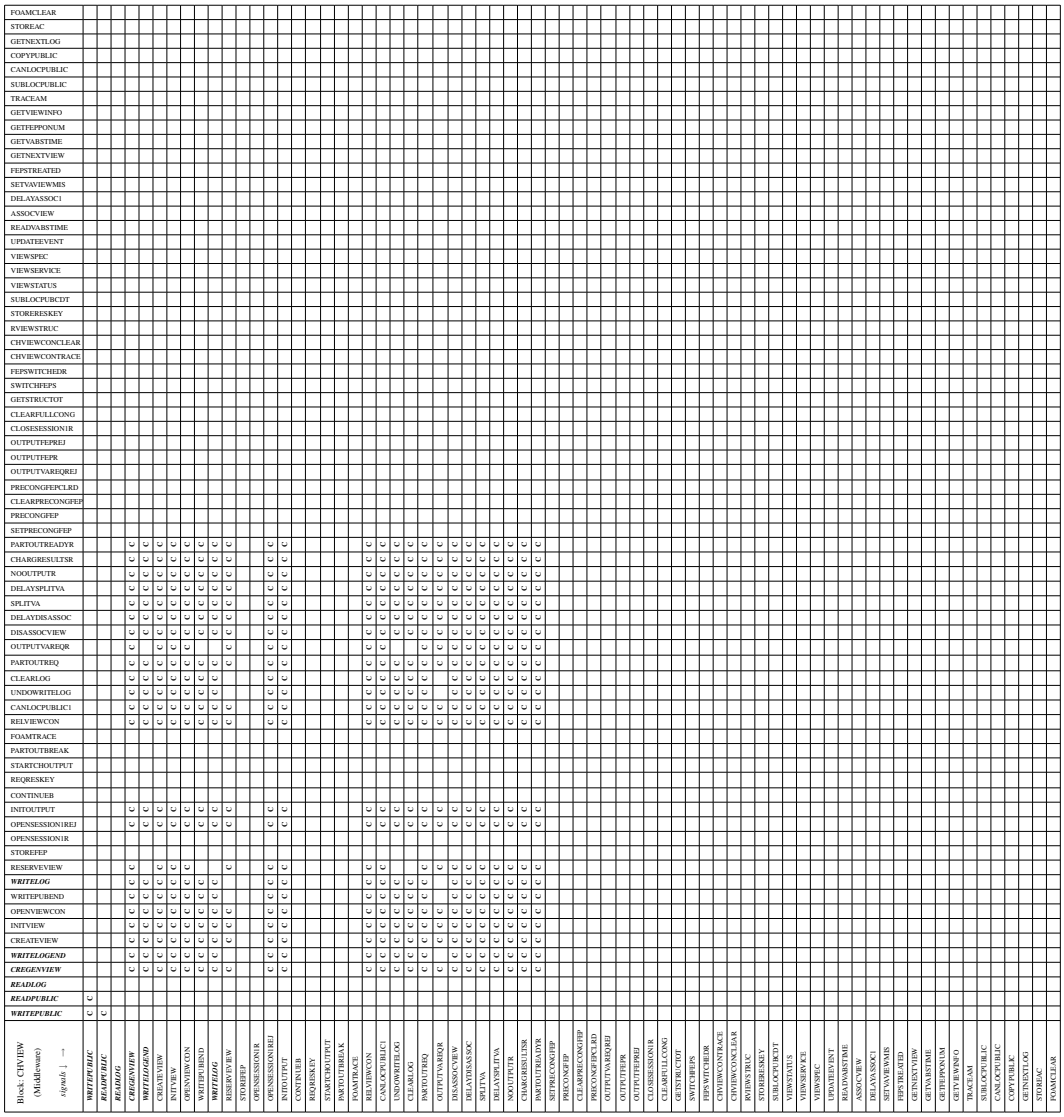

Table C.11: CHVIEW - Potential conflicts (marked with **C**) caused by possible simultaneous access to common variables. Derived from Table C.1.

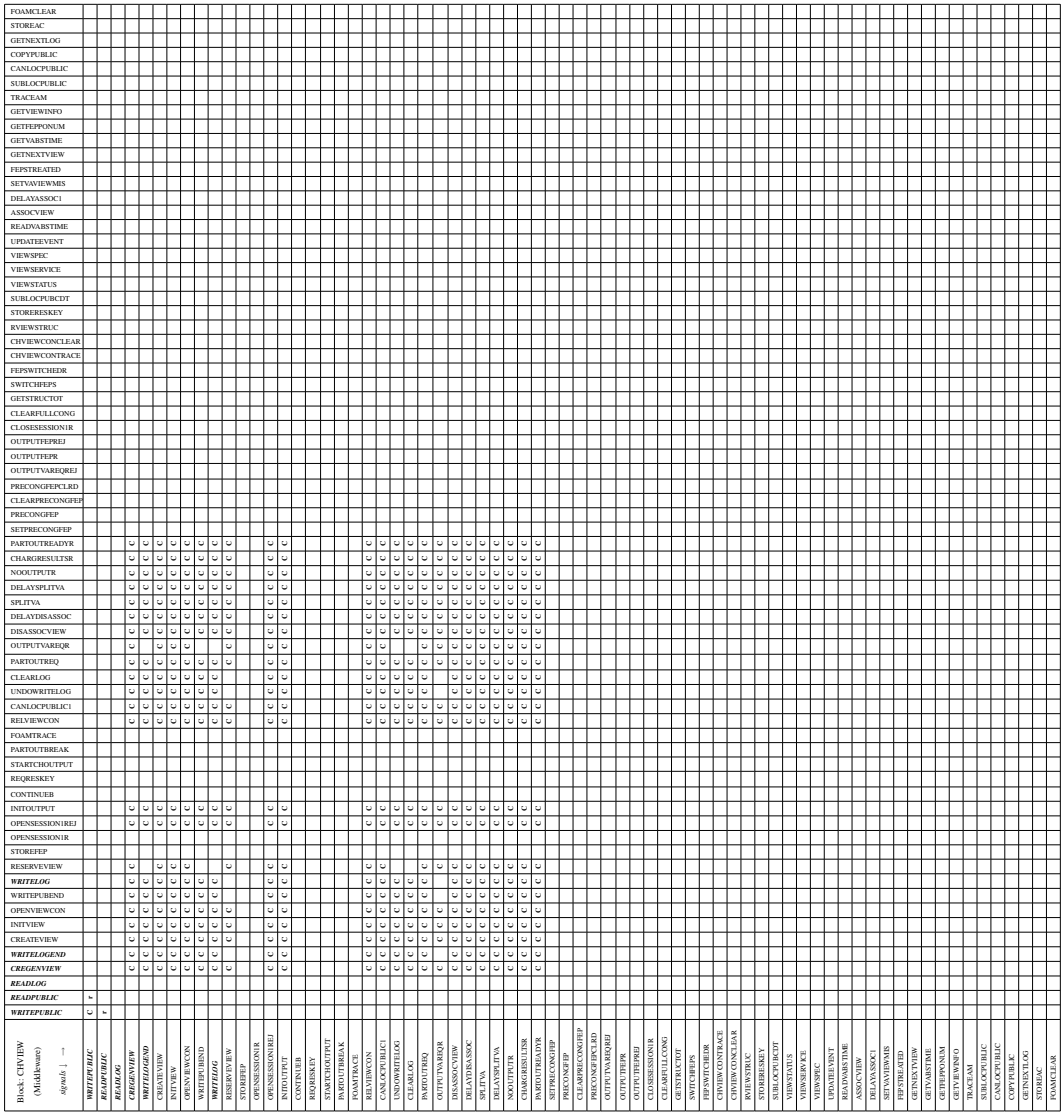

Table C.12: CHVIEW - 'w-optimization' applied on Table C.11. Removed conflicts are marked with **r**.

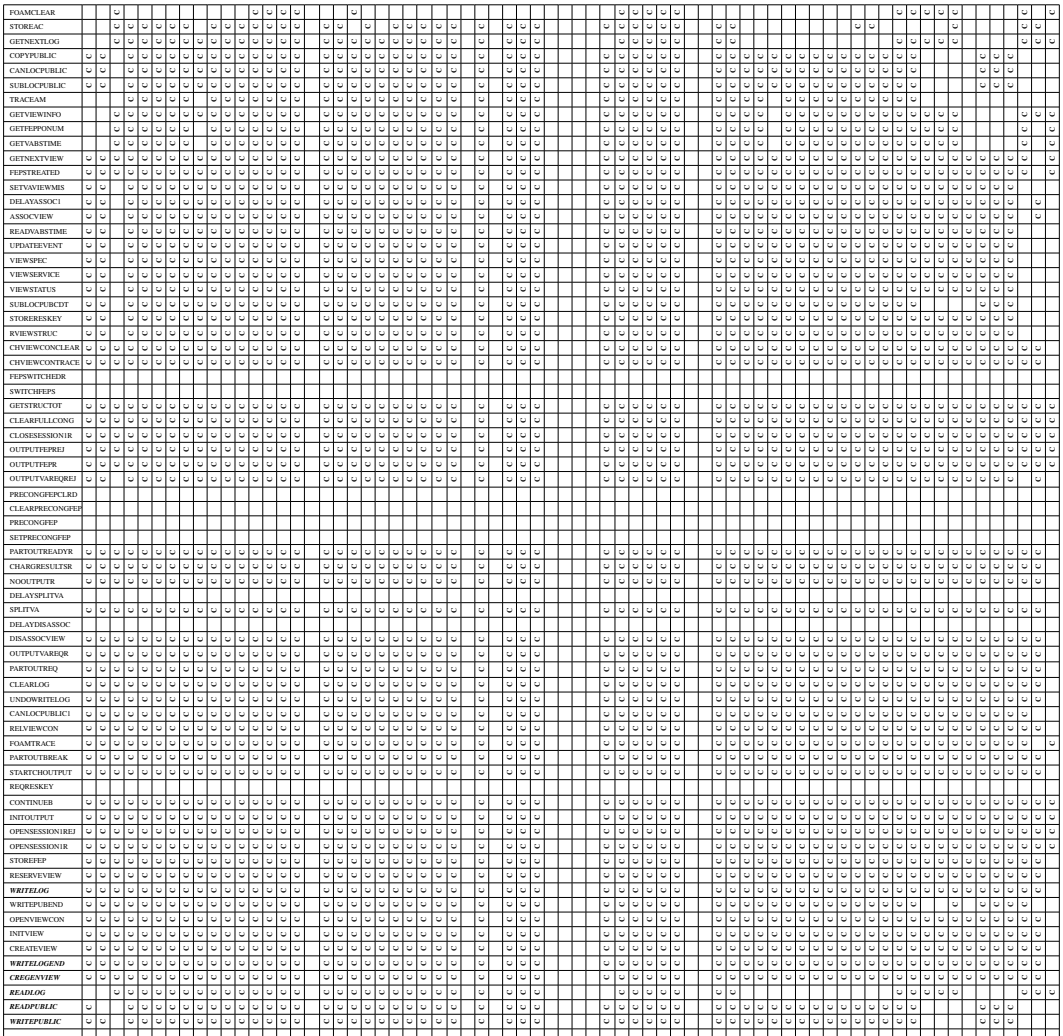

Table C.13: CHVIEW - Potential conflicts in the file variables.

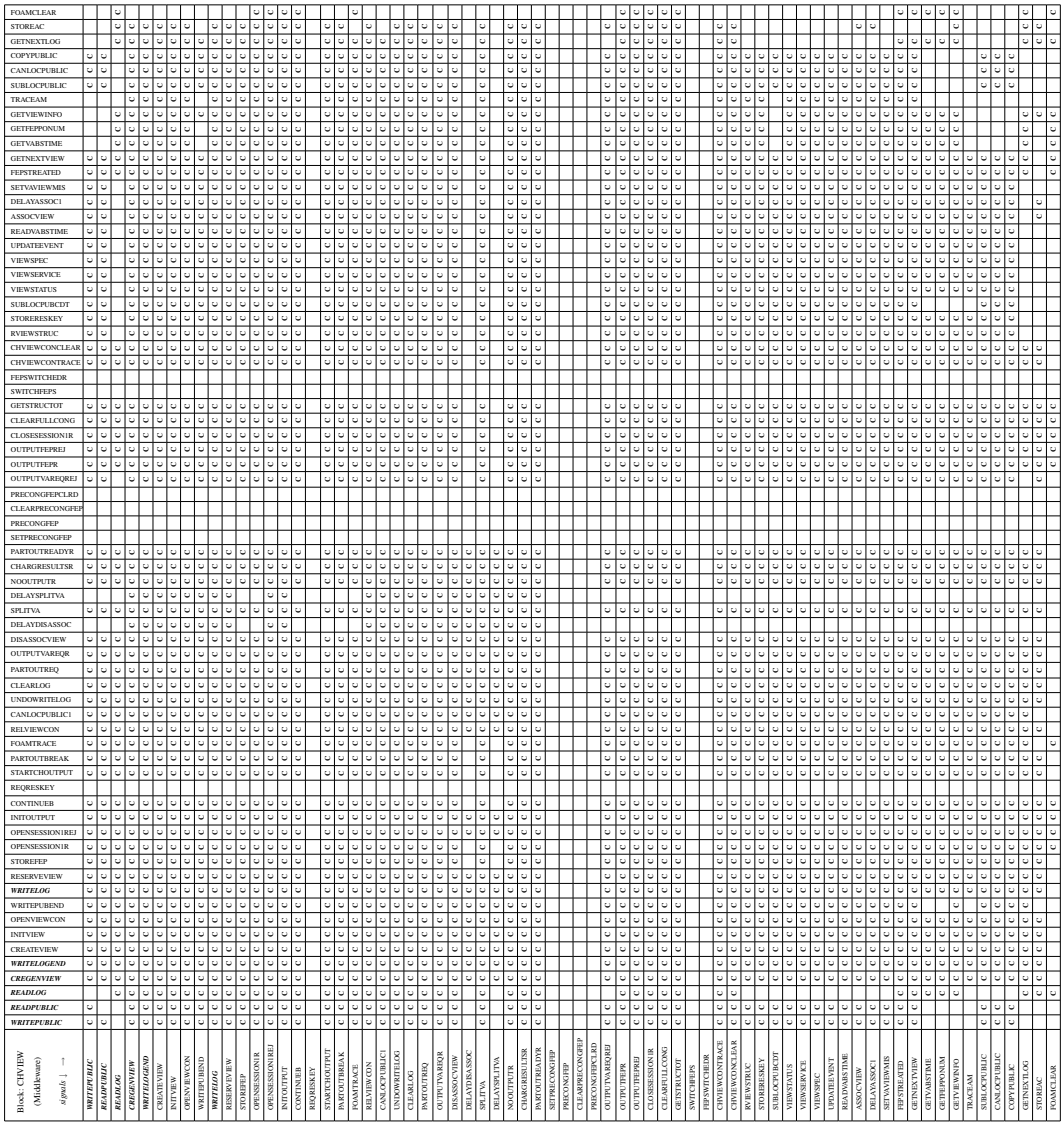

Table C.14: CHVIEW - A safe upper bound of the number of conflicts achieved by combining Table C.12 and C.13.

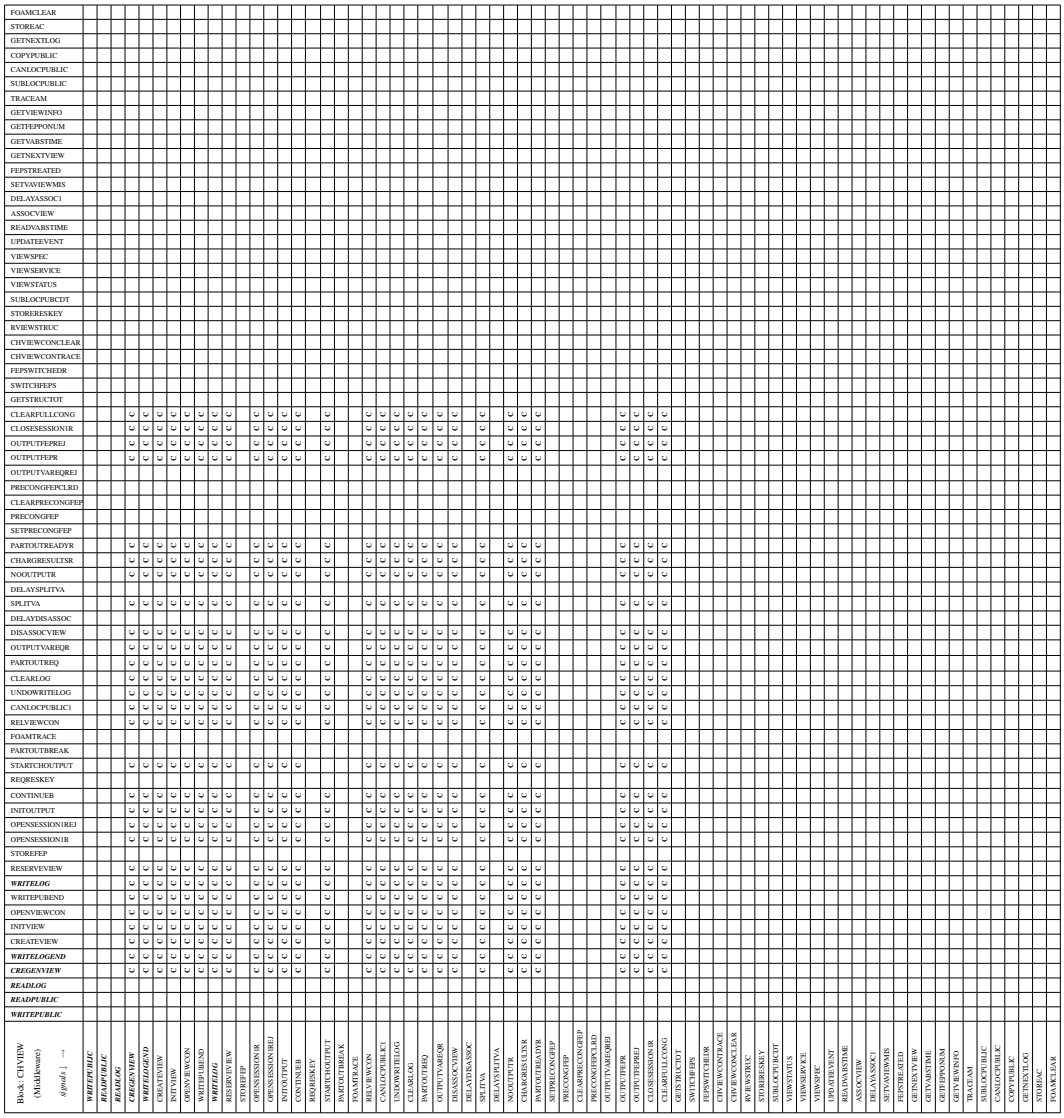

Table C.15: CHVIEW - A tightened upper bound is achieved by assuming a protected SEIZE operation.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF		**C**	**C**	**C**	**C**			
GETBUFABSADR	**C**		**C**	**C**	**C**			
READBSIZE	**C**	**C**		**C**	**C**			
ALLCOMBUF	**C**	**C**	**C**	**C**	**C**	**C**	**C**	
CBFLCONNECT	**C**	**C**	**C**	**C**		**C**	**C**	
GETCOMBUFREF				**C**	**C**		**C**	
RELCOMBUF				**C**	**C**	**C**	**C**	
READCBSTATE								

Table C.16: LAD - Potential conflicts in the common variables.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF		**r**	**r**	**r**	**r**			
GETBUFABSADR	**r**		**r**	**r**	**r**			
READBSIZE	**r**	**r**		**r**	**r**			
ALLCOMBUF	**r**	**r**	**r**	**C**	**r**	**r**	**C**	
CBFLCONNECT	**r**	**r**	**r**	**r**		**r**	**r**	
GETCOMBUFREF				**r**	**r**		**r**	
RELCOMBUF				**C**	**r**	**r**	**C**	
READCBSTATE								

Table C.17: $WriteBeforeRead$-conflicts removed from Table C.16.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF								
GETBUFABSADR								
READBSIZE								
ALLCOMBUF				**C**	**C**	**C**	**C**	**C**
CBFLCONNECT				**C**	**C**	**C**	**C**	**C**
GETCOMBUFREF				**C**	**C**		**C**	
RELCOMBUF				**C**	**C**	**C**	**C**	**C**
READCBSTATE				**C**	**C**		**C**	

Table C.18: LAD - Potential conflicts in the file variables.

Block: MFM (OS) *signals* ↓ →	FLSTARTUNRQ	***FLSTARTCORQ***	FLPARTRQ	FLPARTEXTRQ	***FLLEAVERQ***	FLJOINRQ	FLUNJOINRQ	FLSAVERQ	FLUNSAVERQ	FLABORTRQ	FLRELEASERQ	INQRESTFLRQ	RESTARTFLRQ	MFMOPERATER	CPBREAKR	INITFLERROR	FLPROTECTRQ	CHECKJOINFID	INQSTATFLRQ	FLERRORCREATE	***FLAUDITRQ***	EVENTMESSAGEA	MFMREADR	FLERRORINFR	FLUPDATEDUMP	FLERRORRQ
FLSTARTUNRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLSTARTCORQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLPARTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLPARTEXTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLLEAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLUNJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLSAVERQ	C	C	C	C	C	C	C		C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLUNSAVERQ	C	C	C	C	C	C	C	C		C	C	C	C	C	C	C	C			C	C	C	C			C
FLABORTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
FLRELEASERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
INQRESTFLRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C					C	C	C	C			
RESTARTFLRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C					C	C		C			
MFMOPERATER	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
CPBREAKR	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
INITFLERROR	C	C	C	C	C	C	C	C	C	C	C			C	C		C					C				C
FLPROTECTRQ	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C									C
CHECKJOINFID																										
INQSTATFLRQ																										
FLERRORCREATE	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C						C		C			
FLAUDITRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C					C	C	C	C			
EVENTMESSAGEA	C	C	C	C	C	C	C	C	C	C	C	C		C	C	C					C					
MFMREADR	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C					C	C		C			
FLERRORINFR																										
FLUPDATEDUMP																										
FLERRORRQ	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C									

Table C.19: MFM - Possible conflicts in the common variables, with/without optimization.

Block: MFM (OS) *signals* ↓ →	FLSTARTUNRQ	***FLSTARTCORQ***	FLPARTRQ	FLPARTEXTRQ	***FLLEAVERQ***	FLJOINRQ	FLUNJOINRQ	FLSAVERQ	FLUNSAVERQ	FLABORTRQ	FLRELEASERQ	INQRESTFLRQ	RESTARTFLRQ	MFMOPERATER	CPBREAKR	INITFLERROR	FLPROTECTRQ	CHECKJOINFID	INQSTATFLRQ	FLERRORCREATE	***FLAUDITRQ***	EVENTMESSAGEA	MFMREADR	FLERRORINFR	FLUPDATEDUMP	FLERRORRQ
FLSTARTUNRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLSTARTCORQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLPARTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLPARTEXTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLLEAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLUNJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLSAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLUNSAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLABORTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLRELEASERQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
INQRESTFLRQ	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
RESTARTFLRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
MFMOPERATER																										
CPBREAKR	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
INITFLERROR																										
FLPROTECTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
CHECKJOINFID	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
INQSTATFLRQ	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
FLERRORCREATE	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
FLAUDITRQ	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
EVENTMESSAGEA																										
MFMREADR	C	C	C	C	C	C	C	C	C	C	C	C	C		C		C	C	C	C	C		C			
FLERRORINFR																										
FLUPDATEDUMP																										
FLERRORRQ																										

Table C.20: MFM - Possible conflicts in the file variables.

Block: MFM (OS) *signals* ↓ →	FLSTARTUNRQ	***FLSTARTCORQ***	FLPARTRQ	FLPARTEXTRQ	***FLLEAVERQ***	FLJOINRQ	FLUNJOINRQ	FLSAVERQ	FLUNSAVERQ	FLABORTRQ	FLRELEASERQ	INQRESTFLRQ	RESTARTFLRQ	MFMOPERATER	CPBREAKR	INITFLERROR	FLPROTECTRQ	CHECKJOINFID	INQSTATFLRQ	FLERRORCREATE	***FLAUDITRQ***	EVENTMESSAGEA	MFMREADR	FLERRORINFR	FLUPDATEDUMP	FLERRORRQ
FLSTARTUNRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLSTARTCORQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLPARTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLPARTEXTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLLEAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLUNJOINRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLSAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLUNSAVERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLABORTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
FLRELEASERQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
INQRESTFLRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C			C	C	C	C			
RESTARTFLRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C	C	C	C	C		C			
MFMOPERATER	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C			C
CPBREAKR	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C			C
INITFLERROR	C	C	C	C	C	C	C	C	C	C	C			C	C		C					C				C
FLPROTECTRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C			C
CHECKJOINFID	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
INQSTATFLRQ	C	C	C	C	C	C	C	C	C	C	C		C		C		C				C		C			
FLERRORCREATE	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C				C		C			
FLAUDITRQ	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C	C	C	C	C	C	C			
EVENTMESSAGEA	C	C	C	C	C	C	C	C	C	C	C	C		C	C	C					C					
MFMREADR	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		C	C	C	C	C		C			
FLERRORINFR																										
FLUPDATEDUMP																										
FLERRORRQ	C	C	C	C	C	C	C	C	C	C	C			C	C	C	C									

Table C.21: MFM - A safe upper bound of the number of conflicts achieved by combining Table C.19 and C.20.

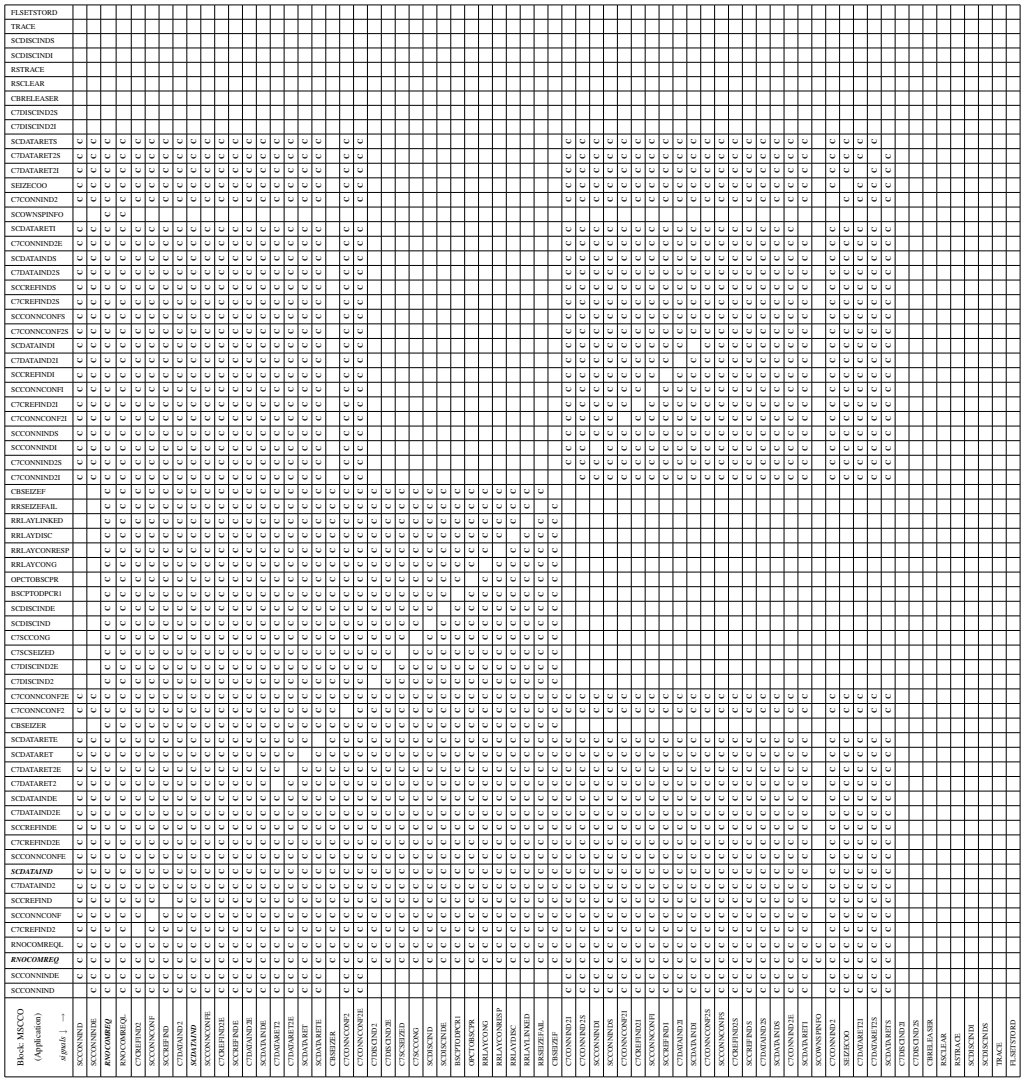

Table C.22: MSCCO - Possible conflicts in common variables, with/without $WriteBeforeRead$- elimination applied.

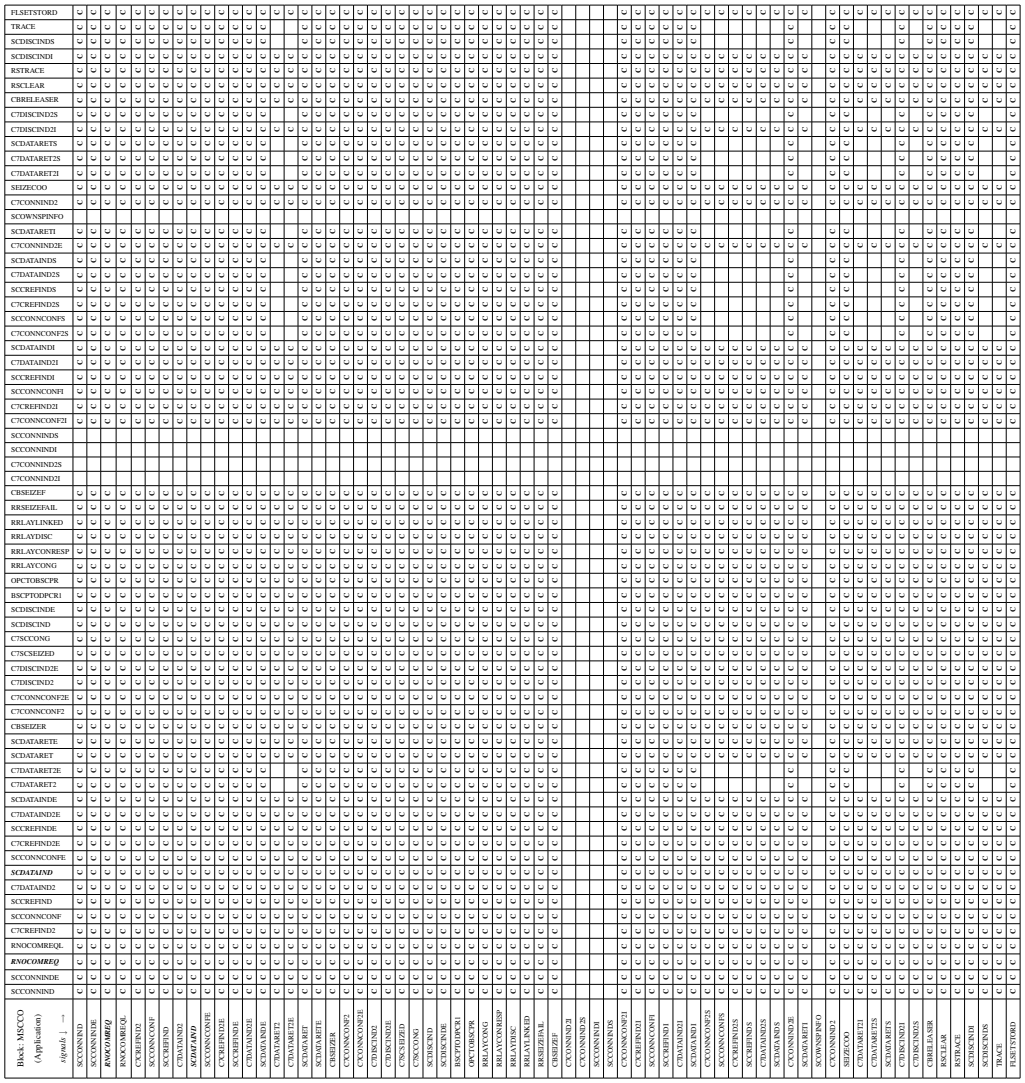

Table C.23: MSCCO - Possible conflicts in the file variables.

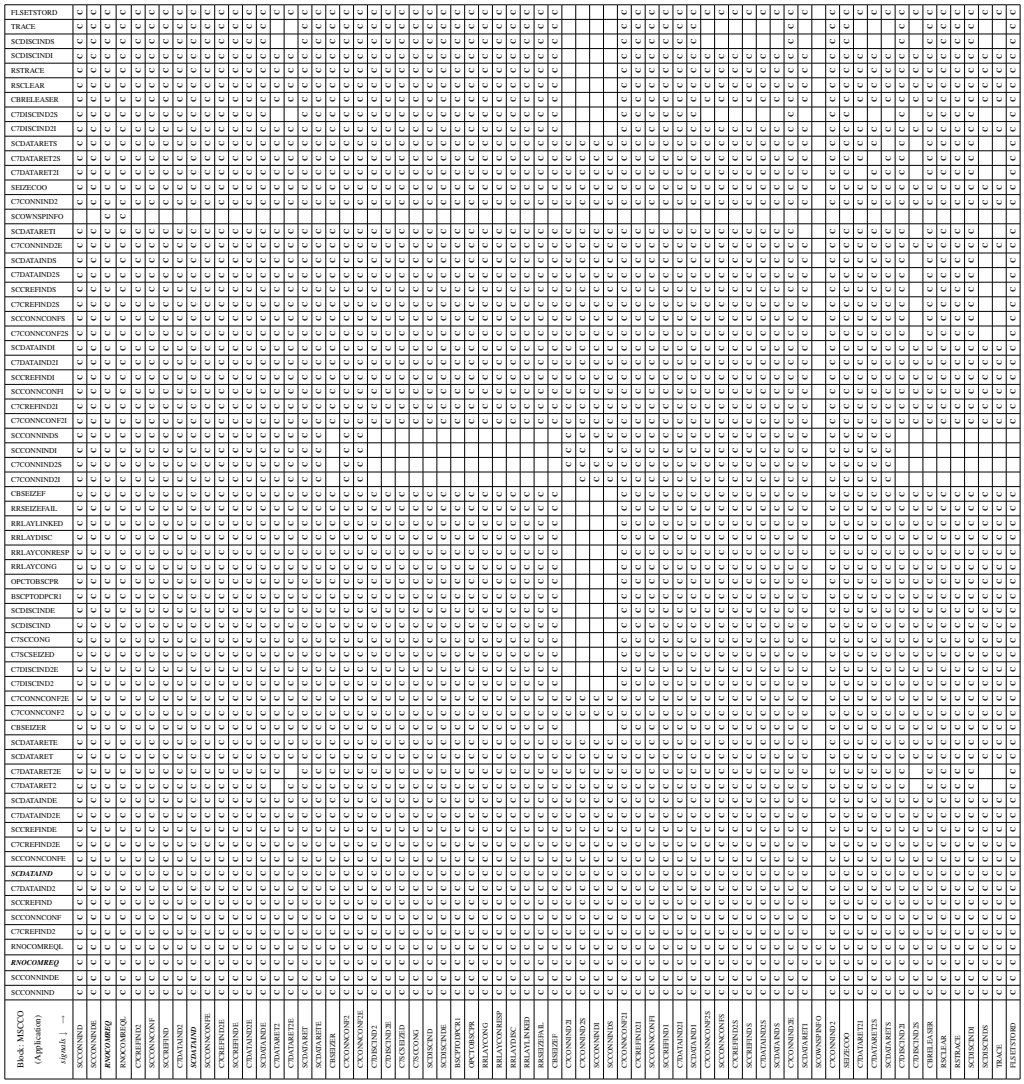

Table C.24: MSCCO - A safe upper bound of the number of conflicts achieved by combining Table C.22 and C.23.