Master Thesis
at the
Department of Information Technology and Electrical Engineering

Distributed Simulation of Dynamic and Fault Tolerant System

AS 2010

Sudhanshu Shekhar Jha

Advisors: Iuliana Bacivarov
Hoesoek Yang
Professor: Prof. Dr. Lothar Thiele

ETH Zürich
2nd March 2011

Abstract

There has been a considerable research in building many real-time multimedia, signal processing applications running concurrently on an MPSoC based embedded systems without application control. To handle the complexity of multiple co-existing and co-executing applications on an embedded multi-core platform (EURETILE) with determinism or in a controller manner, a new framework based on hardware/software co-design is required.

The work done in this thesis defines the specification of multiple applications (specifically designed for a MPSoC platform) as an input to the new framework called Distributed Application Layer (built over Distributed Operation Layer). Along with user-defined application specification, the DAL framework is designed to accept a Finite State Machine (FSM). The user defined FSM is used to control the run-time behavior of co-executing applications based on some external event trigger (adding run-time determinism). An important aspect of the framework is to support an optimal FSM state based mapping of the applications. The new framework is incorporates migration of tasks with varying mapping configuration during state transition. The DAL framework is also designed to provide fault tolerance at application level using double redundancy (duplication of applications) static fault tolerance measure.

To get mapping configurations for application processes on to actual hardware, the DAL framework analyzes the system behavior based on functional simulation. The thesis work also included the extension to the distributed functional simulation used in the DOL framework to quantify multiple application behavior using FSM and statically generated external event trigger.

The functional simulation based on SystemC, show promising results in terms of concurrent multiple application behavior based in FSM. The functional simulation shows promising results in terms of scalability (more number of applications). Addition of mapping behavior on to distributed server not only accelerates the overall system simulation but also assists in the emulating a MPSoC behavior. The library used to support the distributed simulations based on SystemC is particularly suited to distribute functional and untimed or approximate-timed transaction level modeling (TLM) simulations. No modifications have been made to the simulation kernel.

Acknowledgements

First of all I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to work on this master thesis in his research group. I would also like to thank my a advisors, Dr. Iuliana Bacivarov and Dr. Hoeseok Yang for their constant support and guidance in the thesis work. They have always helped me in solving difficult problems during the course of this thesis by providing constant feedbacks and devoting their valuable time for discussions. Without their assistance, this work would never have been possible. Furthermore, I would like to thank my family and my friends for their support and motivation.

Sudhanshu Shekhar Jha, 2011

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Contributions . . . 1
1.3 Related Work . . . 2
1.4 Outline . . . 3

2 Framework **5**

2.1 Overview . . . 5
2.2 Distributed Operation Layer . . . 5
2.2.1 SHAPES . . . 5
2.2.2 SHAPES DOL . . . 6
2.2.3 The Programming Model . . . 7
2.3 SystemC and SystemC Distributed . . . 8
2.3.1 SystemC . . . 8
2.3.2 SystemC Distributed . . . 11
2.4 Distributed Application Layer . . . 16
2.4.1 EURETILE . . . 16
2.4.2 EURETILE DAL . . . 16
2.4.3 The Programming Model . . . 16

3 Dynamic Systems **19**

3.1 Overview . . . 19
3.2 Problem Definition . . . 19
3.3 Multiple Applications . . . 19
3.4 Application Control . . . 22
3.4.1 Dynamic Control Specification . . . 22
3.4.2 Dynamic Application Controller . . . 26
3.5 Task Remapping/Migration . . . 28
3.6 Implementation . . . 31
3.6.1 Multiple Applications . . . 31
3.6.2 Finite State Machine . . . 31
3.6.3 Mapping Modifications . . . 31
3.6.4 Support for Multiple Input/Output Channel Port interfacing in SystemC . 34
3.6.5 Database Update for Remapped/Migrated Processes . . . 35
3.6.6 Database Update for Remapped/Migrated Connections . . . 37
3.6.7 Database Update for Remapped/Migrated Channels . . . 38
3.6.8 Database Update for Process Port Mux/Demux blocks . . . 40

4 Fault Tolerance **43**

4.1 Overview . . . 43
4.2 Problem Definition . . . 43

4.3 Fault Definition and Recovery 43
4.4 Fault Tolerance in DAL 46
4.5 Controller Clones 48
4.6 Implementation 49
4.6.1 Database Update for Cloned Processes 49
4.6.2 Database Update for Clone Connections 50
4.6.3 Database Update for Clone Channels 54
4.6.4 Database Update for Process Port Mux/Demux blocks 56

5 Simulator **57**
5.1 DAL Visitor Overview 57
5.2 Master Controller 58
5.3 Run-time Table for Application Control 60
5.4 Control Message 62
5.5 Slave Controller 64
5.6 Application Task Wrapper 64
5.7 SystemC Application 65

6 Experimental Result **69**
6.1 Simple Applications 69
6.2 Complex Applications 70
6.3 FSM State Transition Time 70
6.4 FIFO Performance 72

7 Conclusion and Future Works **77**
7.1 Conclusion 77
7.2 Future Works 78

A Appendix **81**
A.1 DOL Specification 81
A.1.1 Application and System Specification 81
A.2 DAL Specification 83
A.2.1 Scenario based Applications and System Specification 83
A.3 DAL Naming Convention/Nomenclature 89
A.4 Execution Steps 92

B On-line DAL Example **95**
B.1 DAL Specification 95
B.1.1 Application and System Specification 95

C Presentation **101**

D Acronyms **111**

List of Figures

2.1 DOL Y-chart Approach . . . 7
2.2 SystemC Comparison . . . 9
2.3 The SystemC Scheduler . . . 10
2.4 DOL HDS Visitor . . . 11
2.5 SCD Local State . . . 12
2.6 SCD Master State Chart . . . 13
2.7 SCD Slave State Chart . . . 13
2.8 SCD Request Handling . . . 14
2.9 DOL HDSD Visitor . . . 15
2.10 Modified Y-Chart Approach for DAL . . . 17

3.1 Multiple KPN Applications . . . 20
3.2 Merged Process Network . . . 21
3.3 DOL data parsing and processing design flow . . . 22
3.4 DAL data parsing and processing design flow . . . 23
3.5 A sample FSM for DAL Applications . . . 24
3.6 DAL middle-ware planes . . . 25
3.7 FSM State and Process Liveliness . . . 26
3.8 Control Hierarchy in MPSoC . . . 27
3.9 Application Behavior . . . 29
3.10 FSM and State-based Mapping . . . 30
3.11 DAL entity relationship model . . . 32
3.12 Multiplexer and De-multiplexer . . . 34

4.1 Static fault tolerance setup . . . 44
4.2 Recovery in static fault tolerance . . . 44
4.3 Dynamic Fault Tolerance setup . . . 45
4.4 Recovery in dynamic fault tolerance mechanism . . . 45
4.5 Fault Tolerance in DAL . . . 46

5.1 DAL hdsd work-flow . . . 58
5.2 Master controller work-flow for HDSD visitor . . . 61
5.3 Modified read and write . . . 67
5.4 Control Message Processing at Slave . . . 67

6.1 State Transition Processing Time . . . 73

A.1 Process Network mapping in DOL Framework . . . 81
A.2 Process Network mapping in DAL Framework . . . 84

B.1 DAL working scenario . . . 96
B.2 DAL tool chain . . . 100

1

Introduction

1.1 Motivation

In recent times, there has been a considerable surge and push towards building real-time multi-media, signal processing and embedded systems over a MPSoC (multi-processor system-on-chip) architecture (both homogeneous and heterogeneous architecture based on requirement and cost). The multi-processor architecture is preferred over single-processor architecture because of higher performance gain at lower energy consumption and lower heat dissipation.

On the down side, the development and analysis of an application implemented for a MPSoC is difficult. One of the main challenges for software development for MPSoC platform is to keep all core and hardware accelerators busy to achieve maximum throughput. Basically, this can be achieved by mapping the different processes of an application to the different cores. This also assists in achieving a load-balanced system. This approach, static in nature, is limited to applications where information about the process and the corresponding resource demand is known at compile-time (without any deviation). However in recent times, there has been a need for a system that would facilitate dynamic behavior of applications. These applications may be active or inactive during the normal execution of the system. Further, the processes can have aperiodic non-uniform execution behavior and can still demand for resources (memory or processors/hardware accelerators or bus). This would result in leading to an unpredictable hardware utilization which may even lead to non-uniform workload distribution at a given time.

Due to scope of real-time applications in the real world, the system is required to support recovery from hardware or software faults to avoid system reboot or replacement or objective loss; hence dynamic in nature.

1.2 Contributions

This theme of the project is to extend an existing MPSoC design-flow called Distributed Operation Layer (DOL) [1, 2] toward the Distributed Application Layer (DAL) to introduce dynamism. The overall goal is segmented into three tasks.

The goal of the first task is to define a specification environment for supporting dynamic control and concurrent execution of multiple applications. The application specification used in DOL

is modified to capture the proposed dynamic characteristics. In addition to this, a co-ordinated structure for applications is defined to support dynamic properties.

The scope of first task also include the method to define the multiple mapping options for co-existing and co-executing applications tasks distributed on the hardware. The DAL proposes FSM state based application mapping and switching between mapping configurations at run-time. This is termed as task remapping/migration. In DAL, the dynamic mapping can be tested at system-level using abstract hardware platform and at functional-level (application functionality level).

The second task defines structure to handle system faults and implement the fault-tolerant behavior for applications and processors using re-mapping and redundancy/back-up mechanism. This is one of the requirements of the DAL Framework. At system-level, fault-tolerance is achieved via re-mapping. When a fault is detected in the system (e.g., faulty processor), back-up processes is activated to prevent the system from stalling.

The third task generate a functional simulation environment with the above two DAL requirements. A distributed SystemC simulation was creates to emulate the MPSoC behavior. The application tasks are mapped onto these SystemC applications and executed concurrently using FSM and statically generated events.

1.3 Related Work

The need for specification, analysis and synthesis of high performance signal processing and streaming multimedia applications is of utmost importance for MPSoC based architecture. The representation of applications using Process Networks and Data Flow Models is the most obvious choice. A wide variety of Data Flow models/system support advanced analysis techniques in addition to automatic synthesis for applications. The Data Flow Systems like Synchronous Data Flow (SDF) [3] and extensions to SDF, such as cyclo-static dataflow [4], support such properties. The SDF and its variants can also be used to expresses as static Data Flow Model; the order of firing of a task is known at compile-time or deterministic. On the other side, the dynamic Data Flow Systems like Kahn Process Networks (KPNs) [5], expresses applications with unknown schedule; the firing order is not known at compile-time hence non-deterministic. Many of the existing run-time environments for generating high performance parallel application over MPSoC commonly use simulation frameworks [1, 6, 7, 8, 9, 10, 11].

The analysis of streaming-based real-time embedded applications on MPSoC is getting more focused on run-time existence of multiple applications on the MPSoC cores. The analysis of Multiple applications on an MPSoC have be described in [12, 13, 14, 15, 11]. The most common approach followed in either of the above works is to include dynamic arbitration like admission control [13] or admission manager [15] or run-time implementation of Heavy function [11] to start an application. There is no inclusion of application control based on event.

The demand to combine the properties of non-deterministic application behavior and event-based communication has lead to modification/extensions to existing pure Data Flow Models [16, 17]. The author of Probe [16] extends the data flow model to probe channels for data packets. This modification assist in higher expressiveness of the data flow model at the cost of violation of determinacy property. On the other hand, the designers of Y-API [17] introduced a *select* clause on top of existing KPN clause (blocking *read* and non-blocking *write*) [5]. The Application programmer needs to select the channel during communication phase. This adds determinism to the existing model as the user programmer has an idea of order of firing of processes/task. In Ptolemy [18], a framework is defined to fuse data flow model of computation with event-based model of computation. This adds a great amount of synchronization overhead (at implementation level). The FunState [19] defines a model with functions driven by state machines; data flow independent control flow. The *charts [20] was one of the initials works describing a separate hierarchical finite state machine control model and data flow models.

The approach in [21] focuses on reconfiguration as a particular kind of event handling (control flow). They define *quiescent points* in the execution where reconfiguration is allowed. They also propose the use of FIFO channels for events or parameters communication and to divide input ports in streaming input ports and parameter input ports. The work is primarily focused on analysis and ability to schedule reconfigurable SDF graphs.

The approach defined in [22] use a FSM per actor to model control on an actor. Inclusion of FSM per actor would be ideal if the application comprises of single process instead of a process network. Addition of FSM per actor/process would be redundant if the application control is considered at system-level. The authors also try to abstract multiple channels for efficiency and easy of synthesis.

In Reactive Process Networks [23], KPNs [5] are used a data flow model unlike SDF in [21]. In RPNs, a unified model is used to represent data-oriented data flow and control-oriented data flow. In the paper, the authors defines this model and a set of formal operational semantics allowing description of an application at different abstract levels of computation. The model also incorporates the possibility of dynamically changing the structure of the Process Networks as opposed to reconfiguration with altered parameters [21]. This model is used as a basis for this master thesis.

The concept of State based Task Migration (optimal mapping) adopted in this work has not been discussed in the available literature*. Conceptually, Task Migration on MPSoC is discussed in [24, 25, 26]. In [26], the authors discuss the task migration over heterogeneous MPSoC cores using relationship across instruction sets (core-specific). This approach is more general and would not scale across varying hardware platforms. The concept of Task Migration referred to in [25], is basically due to on-line movement of process/task to other core due to processor/core overload. Various type of general Task Migration strategies is discussed in [24].

Different class of reliability models mentioned in the paper [27] highlight various fault tolerance model that are incorporated in digital systems. The static reliability models in this paper aptly describe the redundant processor model with higher performance gain at higher silicon cost.

1.4 Outline

The remaining part of this work is distributed over six chapters (excluding the current chapter). The Chapter 2 defines the Framework used for the successful completion of the thesis. It briefly describes the DOL Framework [28] and its key features. The chapter also describes the libraries used to setup distributed functional simulation environment. This chapter also highlights the theme of EURETILE [29] and DAL Framework.

The Chapter 3 describes in detail the first requirement of DAL middle-ware. The chapter highlights the concept behind dynamism and corresponding implementation details. It also describes the methodology used for supporting task remapping/migration for multiple co-existing mapping strategies.

The feature of system-level and application-level fault tolerance is described in Chapter 4. This chapter also includes the implementation details incorporated for fault tolerant mechanism.

The implementation aspects of SystemC based simulator is mentioned in Chapter 5. The implementation aspects described in this chapter is valid for DAL hdsd visitor and can be incorporated to other visitors as well. The implementation details included in this chapter facilitate the DAL code-generation phase and run-time/dynamic behavior of applications. It also includes the visitor specific constrains encountered during implementation phase.

The experimental setup created to test the DAL requirements is illustrated in Chapter 6 along with results. The chapter begins by describing simple application scenario and then proceeds to

*The time frame mentioned here refer to the time since the beginning of this thesis work.

complex scenarios with higher number of application tasks. The chapter concludes with some performance statistics gathered during the experimentation phase.

The report ends with a chapter 7 on conclusion of work done and possible future works to extend or reuse the implementation details.

2

Framework

2.1 Overview

This chapter deals with the details of underlying models or frameworks used for the development of the dynamic multiple application model for a MPSoC platform.

2.2 Distributed Operation Layer

The Distributed Operation Layer (DOL) is a design flow for model-driven development [30] of multiprocessor streaming applications and has been developed at the Computer Engineering Group at ETH Zürich. This platform is developed in the context of a European Framework Programme 6 research project called Scalable Software/Hardware Architecture Platform for Embedded Systems (SHAPES) [31].

2.2.1 SHAPES

The Shapes project dealt with the hardware-software co-design of a tiled multiprocessor. Exploiting the raw performance of the SHAPES multiprocessor architecture without resorting to time-consuming device-level programming was the main challenge from a software perspective. The proposed heterogeneous tiles consist of various cores, for instance a RISC processor, a very long instruction word (VLIW) DSP, a distributed network processor and on-tile memories and peripherals [31]. In addition to performance, scalability was also one of the most important aspect of SHAPES project. In other words, an application should be portable on different SHAPES hardware architecture (with minor tweaks); it should be possible to map an application onto a given architecture with varying number of tiles. Table 2.1 shows the SHAPES target for the range of scalability [32].

The tiled architecture approach is advantageous in terms of performance and efficiency, however it is difficult for an application to fully utilize its capabilities. For instance, there may be long delays between tiles due to spacial location, overloaded communication resources or the application design due to lower degree of parallelism. In all these cases, the architectural resources are under-utilized. Therefore the system software should ensure that the applications are executed

efficiently on the SHAPES hardware, with minimal application programmer effort. The two key points taken into consideration for handling this requirement are mentioned below:

- As the architecture is intrinsically designed to be highly parallel, the application should also exhibit highly parallel properties. The application programmer must be able to fully expose the algorithm parallelism to the SHAPES platform. Hence, the conventional way of designing an application cannot be adopted. The information about the algorithmic structure must be preserved by the SHAPES system software even if the application is explicitly written to exposes the parallelism.

- The system software must be fully aware of important architectural parameters and resource constraints like bandwidth, computing capabilities and latencies [32].

Table 2.1: SHAPES Target Architecture and Range of Scalability.

Number of Tiles	Scope of application
<10 tiles	low-end for mass market applications
<200 tiles	classic digital signal processing systems
	(e.g. radar, medical equipment)
>200 tiles	high-end systems requiring massive
	numerical computation

2.2.2 SHAPES DOL

The Distributed Operation Layer is a part of the SHAPES system software environment. This middle-ware is designed to assist the application programmer using the SHAPES platform to find an efficient mapping of the application onto the underlying hardware. This includes mapping of application tasks onto computation resources, as well as the mapping of communication links onto communication resources. To ensure efficient utilization of the parallel hardware resources, the application programmer should follow a set of rules and use a predefined set of interfaces and a strict programming model. In accordance to the DOL programming model, Section 2.2.3, an application may consist of several *processes/tasks* interconnected with *FIFO channels*; built as a process network. There is no existence of shared memory between *processes/tasks*, hence inter-process/task communication is done using *FIFO channels*.

For instance, consider a specific algorithm implementation on the SHAPES platform. The application programmer first creates an algorithm to extract the task-level parallelism using the processes $\mathcal{P}_1 \ldots \mathcal{P}_n$, and the network $\mathcal{N}$ describing inter-process/task connectivity. The programmer explicitly exposes parallelism to the DOL. The exposed parallelism is used for obtaining optimal mapping configuration/strategy. As the extraction of parallelism is done by the programmer, an in-depth knowledge of the application domain is required.

The DOL Framework facilitates creation of run-time environment (termed as visitor in DOL Framework) for process networks located on the same or on different processors. The following run-time environments are supported for the different multiprocessors:

- Atmel Diopsis 940: For SHAPES, a custom multiprocessor Operating System was developed for the Atmel Diopsis 940 called DNA-OS.

- MPARM: For the execution of real-time applications MPARM [33, 34] is used. It provides support for the RTEMS [35] Operating System [36]. The RTEMS OS provides a full-fledged preemptive scheduler and message queues. The application process networks are implemented on top of it [37].

- Cell Broadband Engine: A run-time environment for process networks is also supported for Cell Broadband Engine architecture. For lower run-time overhead and memory usage, the run-time environment is based on a lightweight protothread [38] implementation and windowed FIFO implementation [39].

2.2.3 The Programming Model

The SHAPES programming model is similar to the Y-API programming interface [40], Figure 2.1, with an adoption of an extension of the Kahn Process Network model [5]. The Kahn Process Network is a model for parallel computation, where a program consist of a number of *processes/tasks*, connected by *FIFO channels*. Each process is a usual sequential program extended with a communication interface consisting of two functions: the *wait($\mathcal{U}$)* function, which performs a blocking read access to the specific channel $\mathcal{U}$, and the send $\mathcal{I}$ on $\mathcal{W}$ statement, which writes the variable $\mathcal{I}$ to the channel $\mathcal{W}$. While the *wait* function blocks the process until enough data is available on the *channel*, nothing can prevent a process from sending data over a channel. The processes therefore communicate via infinitely large FIFO queues. The Kahn Process Network has two important properties, namely:

- The *processes/tasks* of a KPN are *monotonic*. In other words, only partial information about the input is needed to compute a partial output i.e. the future input is relevant only for future output. This is an important property as it allows parallelism - a *process/task* can start computing before all of its input values have arrived [5].
- The Kahn Process Network is *determinate*. This means that the same input history always produces exactly the same output, independent of the scheduling.

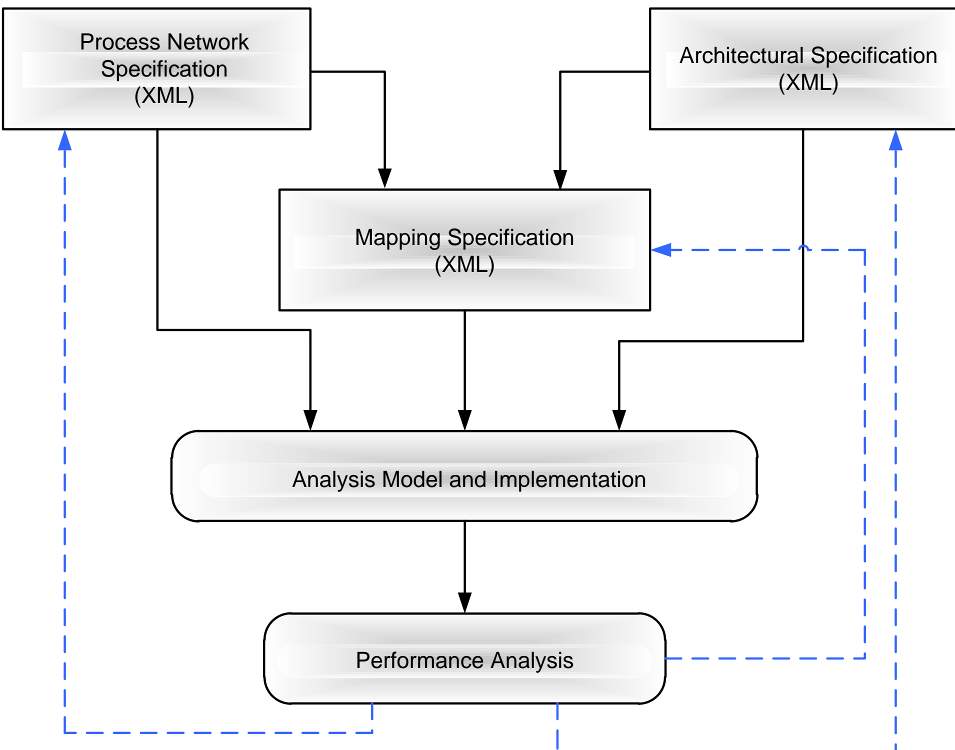


Figure 2.1: Y-chart approach for designing multiprocessors and multiprocessor software for SHAPES platform.

The SHAPES/DOL API has a set of basic communication primitives namely, *DOL_write()* and *DOL_read()*, but differs from the Kahn Process Network model in the following way:

- As infinitely large *FIFO channels* are not realizable, every *channel* is instantiated with a maximum buffer size. This causes the *DOL_write()* function to stall the calling *process/task* if the FIFO queue is full.
- The Kahn Process Network model does not permit to test if the channel $\mathcal{U}$ is empty, before invoking *wait*($\mathcal{U}$) as it can be restrictive in some cases and may cause inefficiency. To avoid this, the SHAPES/DOL API provides two additional functions allowing a read/write test, namely *DOL_rtest()* and *DOL_wtest()*.
- To terminate the simulation gracefully, the function *DOL_detach()* permits the *process/task* to remove itself from the pool of active processes. The simulation ends as soon as all processes are *detached* (no *active processes/tasks*).

The *inter-process/task* communication is facilitated using a list of communication primitives, Table 2.2. Before invoking a blocking read function, a call to the read testing function can be used to check whether the preceding *process/task* has produced the required amount of data. This allows the *process/task* with more than one input port to select a *port* to read at run-time. However, the application programmer has to decide whether the testing functions are actually required by *processes/tasks*. Avoiding the two testing functions would result in a predetermined order of port accesses, hence facilitating a simple process structure and reduction in communication overhead. This would also conform to Kahn Process Network properties mentioned above.

Table 2.2: The DOL Communication Interfaces.

Communication Interface
DOL_read($port, buffer, length, process$) Reads $length$ bytes from $port$ and stores the obtained data in $buffer$. If less than $length$ bytes are available, the calling $process$ is blocked.
DOL_write($port, buffer, length, process$) Writes $length$ bytes from $buffer$ to $port$. If the FIFO connected to $port$ has less than $length$ bytes of free space, the calling $process$ is blocked.
DOL_rtest($port, length, process$) Checks whether $length$ bytes can be read from $port$.
DOL_wtest($port, length, process$) Checks whether $length$ bytes can be written to $port$.
DOL_detach($process$) Detaches the $process$ and prevents the scheduler from firing the $process$ again.

2.3 SystemC and SystemC Distributed

2.3.1 SystemC

The SystemC [41] is a *system design language* that has evolved in response to a pervasive need for a language that improves overall productivity of electronic system designers. SystemC offers higher productivity gains by letting engineers design both the hardware and software components together, as these components would exist on the final system, but at a high level of abstraction. This higher level of abstraction exposes the intricacies and interactions of the entire system.

Thus it enables a better system design trade-offs, better and earlier verification, and over all productivity gains through reuse of early system models as executable specifications (Figure 2.2).

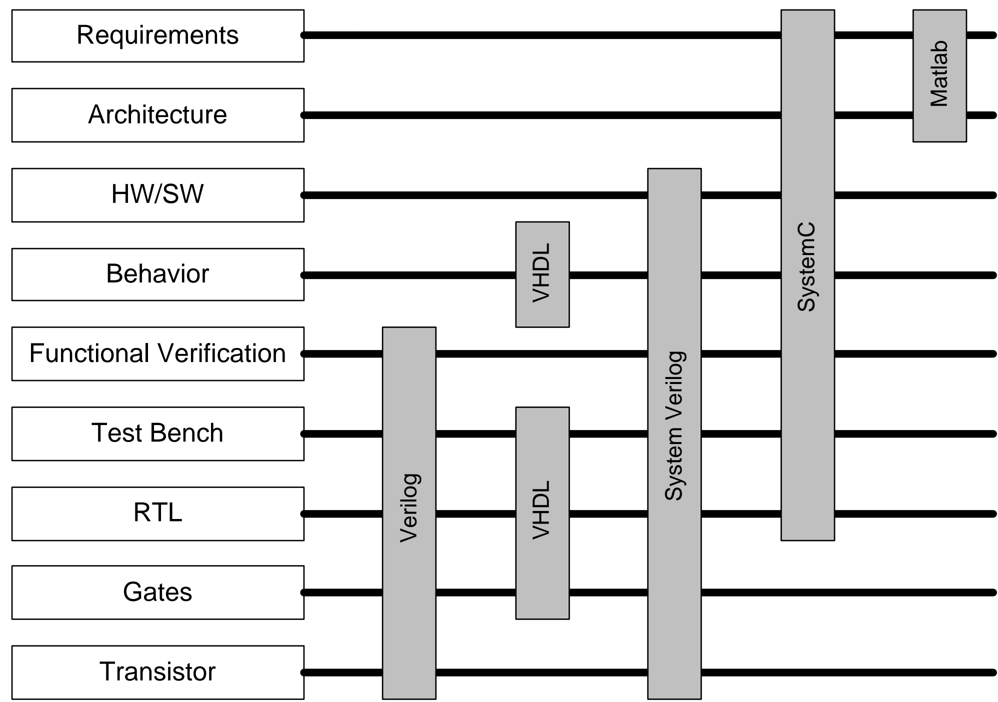


Figure 2.2: Comparison of SystemC with respect to other languages [42].

As the primary goals of SystemC is to enable system level modeling i.e. modeling of systems above the RTL level of abstraction, including systems implemented in software or hardware or a combination of the two. One of the challenges in providing a *system level design language* is that there is a wide range of *design models of computation*, *design abstraction levels*, and *design methodologies* used in system design. To address these challenges in SystemC, a small modeling foundation has been added to the language. On top of this language foundation, more specific models of computation, design libraries, modeling guidelines, and design methodologies can be added as per system design requirements.

Strictly speaking, SystemC is not a language, but a class library defined on top of a well established language, C++. The SystemC is coupled with the SystemC Verification Library to provide many of the characteristics relevant to system design and modeling tasks in a single language that are missing or scattered among the other languages. Additionally, SystemC provides a common language for software and hardware.

The SystemC enables design and verification of systems on different levels of abstraction, thereby supporting development of complex systems. The components can be defined based on specification and the inter-component interaction can be analyzed in an early design phase independent of the target system.

The full SystemC Library specification can be obtained in IEEE Standard 1666 [44]. The SystemC library implementation is provided by the Open SystemC Initiative (OSCI) [45]. The main components in SystemC used for modeling a system are *processes* and *channels*. A *process* describes functionality and allows parallel computation. *processes* are executed without interruption until it yields the control to the simulation kernel by returning or waiting for an event. This can be termed as co-operative or non-preemptive multitasking. The OSCI SystemC is designed to execute only one process at any time even if the hardware supports execution of concurrent processes; resulting in under-utilization of computational resources on modern computing systems.

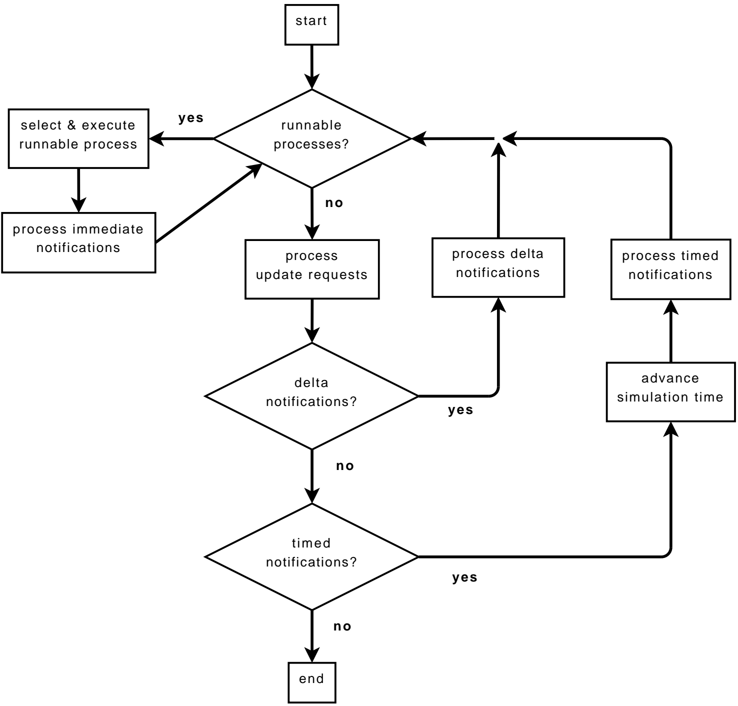


Figure 2.3: The SystemC Scheduler [43].

The *channels* connect two *processes* and allow exchange of data (Primitive SystemC *channels* do not encapsulate any *processes*. Hierarchical channels can be composed of processes and primitive channels and can therefore be more complex). Additionally, SystemC library provides a rich set of predefined channels and custom channels can be defined on top of them or from primitive types.

The SystemC simulation is based on events (*sc_event*). The execution of processes depend on the occurrence of events it is sensitive to. The SystemC Kernel maintains a set different event queues and a set of runnable processes. The Figure 2.3 shows the scheduling behavior of the SystemC simulation.

Using the programming model for SHAPES DOL as in Figure 2.1, the functional simulation using SystemC primitives requires the following steps, resulting in the structure shown in Figure 2.4:

- Parse the XML file to determine the topology of the process network.
- Create the directory structure for proper code organization.
- Create a main file for bootstrapping the process network:
  - Create a *SC_THREAD* that initialized all the actor/task *init()*.
  - For each *actor/task*, create a *SC_THREAD* that repeatedly calls the *fire()* routine depending upon the value of the *_detach* flag.
  - Instantiate *FIFO channels* and bind each port in *actor/task* to its corresponding *FIFO channels* (based in SystemC primitive interface).
- Create a *Makefile* to facilitate the build process of the functional simulation.
- Execute the binary.

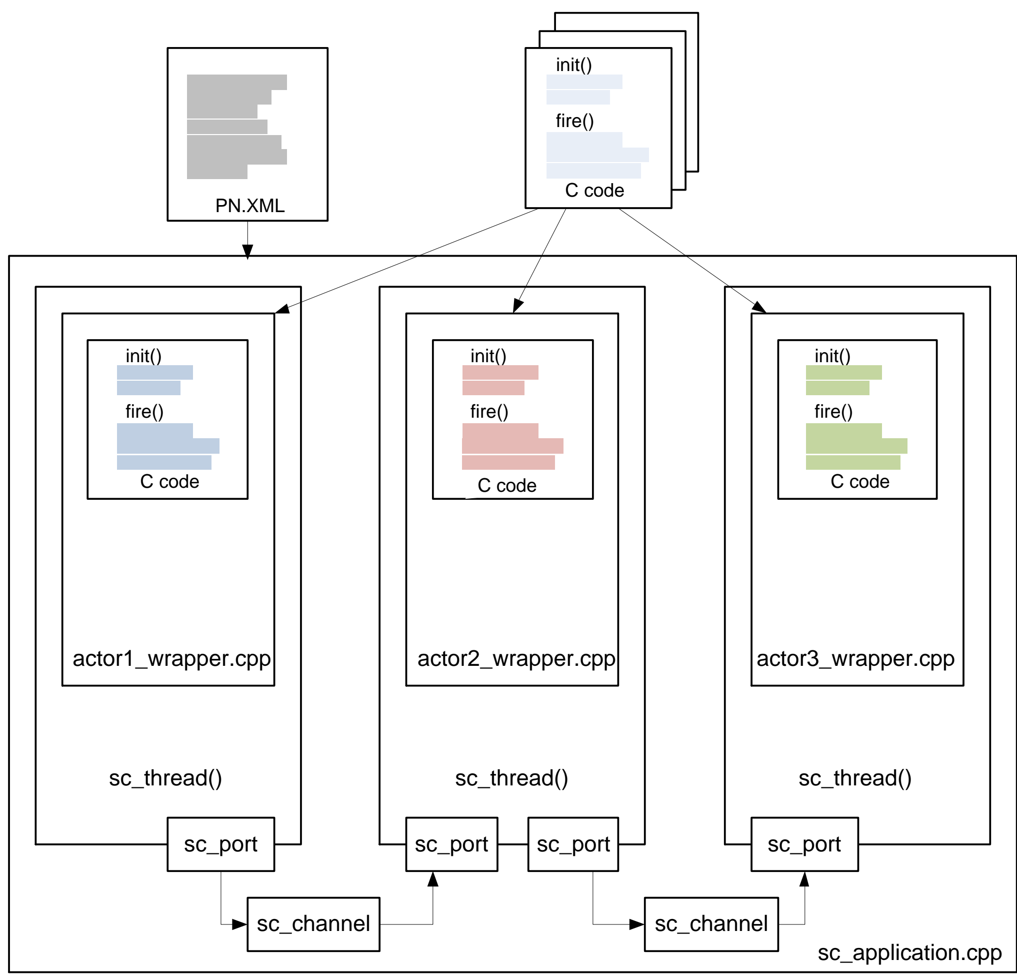


Figure 2.4: Structure of the functional simulation of a process network in DOL based on the SystemC library (uni-processor model).

2.3.2 SystemC Distributed

The SystemC has been designed for single host simulation only. The library does not provide a means to distribute the SystemC simulations. The SystemC Distribution (SCD) [46, 43] library is one of the novel techniques of supporting the geographical distribution of SystemC simulations. It is based on OSCI SystemC 2.2.0 [45]. An arbitrary number of Linux systems (predefined) connected by a network can be used to share the simulation workload. This arrangement can be termed as geographical distribution of SystemC simulations. This is particularly important as modern computation systems contain either multiple CPU cores or a distributed computation environment. This library is designed using the standard SystemC kernel. This technique is well suited for distributed functional and approximated-timed Transaction Level Modeling (TLM) simulations.

The idea of SCD is the execution of the simulation kernels in term of *delta cycle*. This can be modeled by calling *sc_start(SC_ZERO_TIME)* repeatedly. After every *delta cycle*, the simulation jumps back to the main function, i.e. *sc_main()*, where communication and synchronization take place. The communication across distributed simulations is setups using standard TCP-IP connection. A basic synchronization model is adopted to manage and control the simulation activity at a global scope, for instance, advancing the simulation time or terminating the simulation. The communication model is designed to be synchronization independent. This ensures that

the simulation can be continued independently and in a non-blocking fashion even if no data has arrived or if the remote simulator is unable to accept data. In other words, the communication model and the synchronization model are completely independent from the SystemC simulation. The simulation proceeds only when control jumps to *sc_start(timeval)* and it is independent of the existence of events. The progress of the simulation is controlled by obtaining the pending event for current *delta cycle* and future time value for the simulation context.

During a distributed simulation, a simulator might be paused or reactivated based on the data availability from a remote simulator. This model is built using a Control State Machine with simulation states as mentioned below:

- *busy* - The simulator has events at the current simulation time and is simulating.
- *idle* - The simulator currently has no events and it is waiting for the simulation time to advance.
- *done* - The simulator has no events at all and is waiting for the simulation to be terminated.

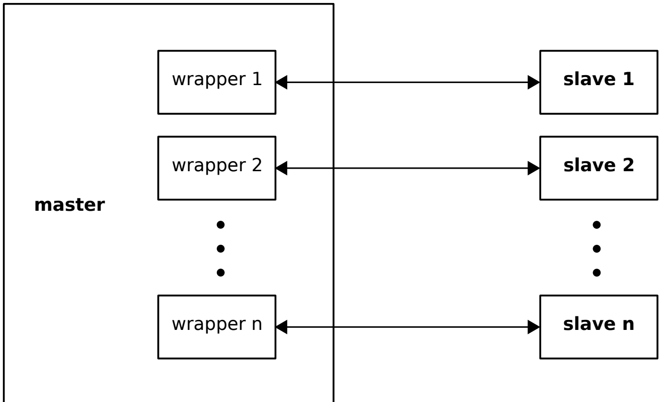


Figure 2.5: Local State of connected Slaves stored in Master in the SystemC Distributed Library [43].

To manage the coherence in global simulation state, a *Master-Slave* approach is adopted. A *Master* simulator is connected to all the *Slave* simulator in the simulation environment. Each *Slave* simulator informs the *Master* about its local simulation state. The *Master* collects these information and advances the simulation time or terminates the simulation depending upon the global simulation state, Figure 2.6 and Figure 2.7. As the *communication* in the distributed simulation environment is built on top of TCP-IP connection using POSIX Sockets [47, 48], the time taken for data transfer across the simulation core is jittered and much higher than local shared memory based communication as native SystemC. This can lead to inaccurate perspective of the *Slaves* local state with respect to *Master* at run-time. Thus, this centralized simulation is governed by the State Machines and *Master* maintains a local copy of the Slave state as shown in Figure 2.5.

The Figure 2.6 and Figure 2.7 show the *Master* and *Slave* Control State Machines using state charts, respectively. The *Master* and *Slave* controllers in the SCD library behave in a similar fashion with an exception that the *Slave* simulator always informs the *Master* about state transitions and its local state. Some of the main states are briefly explained below:

- *init* : All controllers start in the *init* state.
- *idle* : The local state makes a transition to *idle* state when there are no event in the current event queue but is expected to receive an event in future.

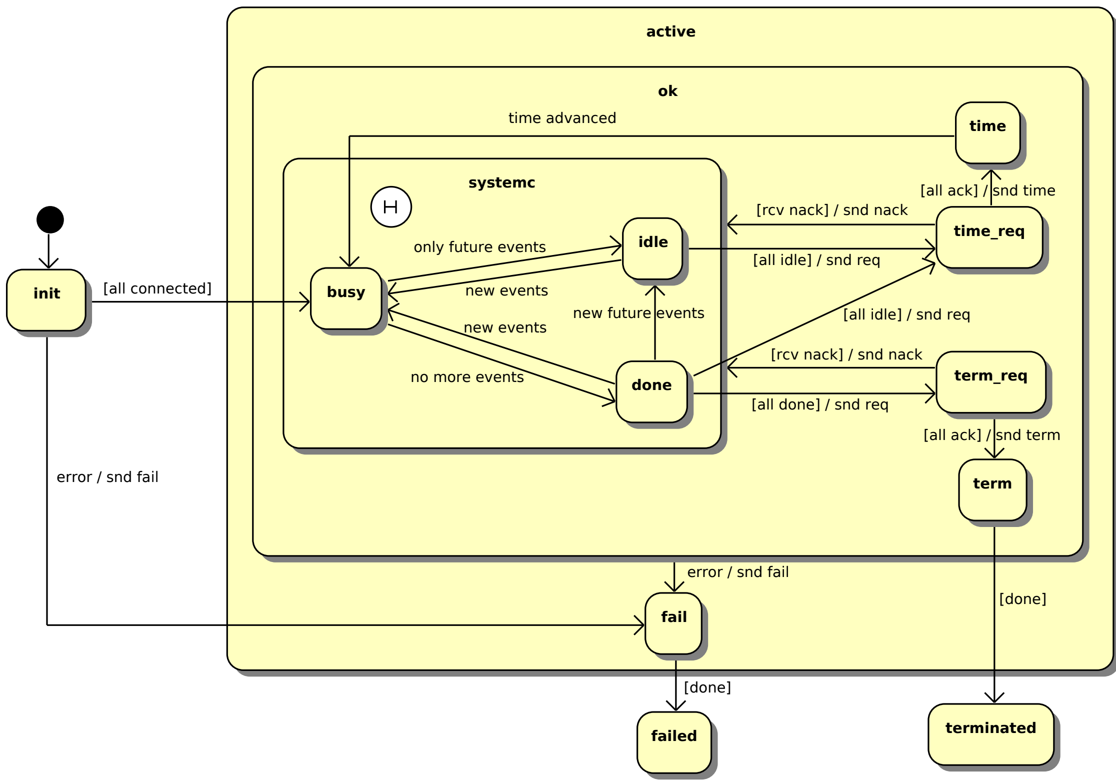


Figure 2.6: State Chart for Master in the SystemC Distributed Library [46].

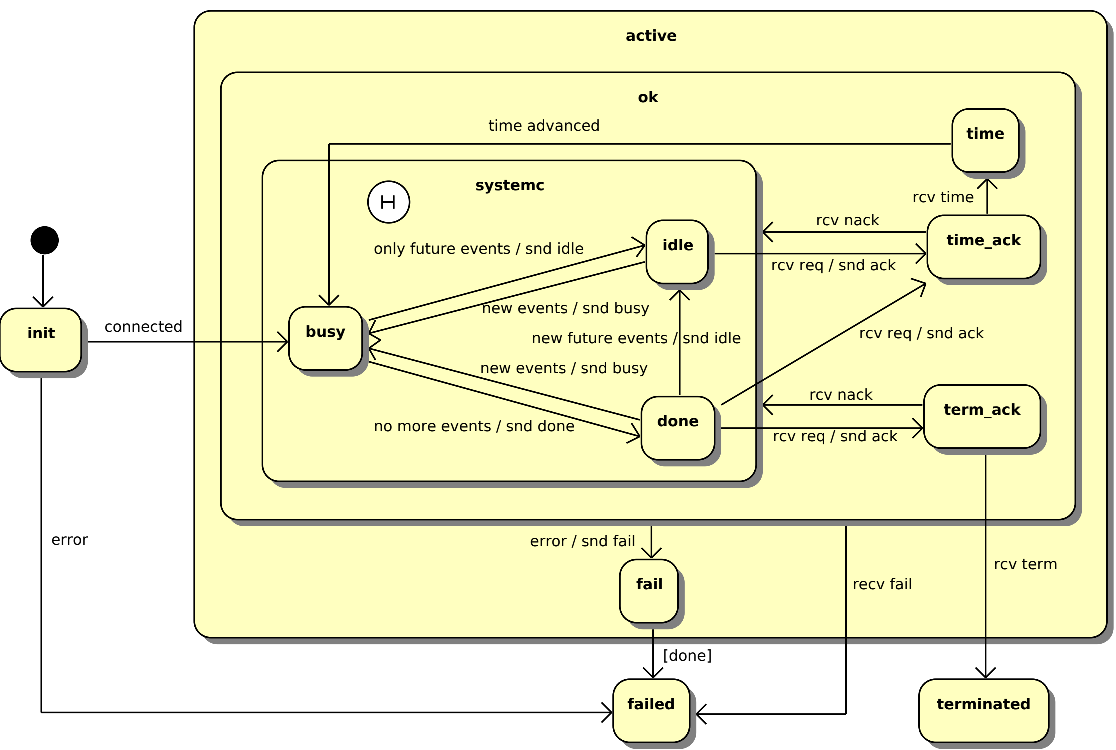


Figure 2.7: State Chart for Slaves in the SystemC Distributed Library [46].

- *busy* : The local state becomes *busy*, once the connection between a Slave and the Master controller is established; the TCP-IP connection using socket return true for *accept()* phase. The Master Simulator waits for all the connection to be in established phase before making a transition to *busy* state.

- *time*: Transition to *time* is triggered if none of the Slaves are in *busy* state. If all Slaves acknowledge the *time* request, the Master determines the time of the next event. It sends a message to all the Slaves to inform them about the advancement in the simulation time. All simulators are moved to the *time* state. As soon as the simulation time has been advanced as per the Master message, the local state of the Slaves are changes to *busy*.

- *fail* : In case of error during simulation, the respective Slave changes its local state to *fail* state. In case of error in the Master simulation, the Master sends a *fail* message to all Slaves and changes its local state to *fail*.

- *failed* : Once the Slave sends the *fail* message to the Master, it moves to *failed* state. Whereas the Master moves to *failed* state once it has sent *fail* message to all the connected Slaves.

- *terminated* : A simulator make a transition to *terminated* state and simulation terminates, if its control state machine leaves the active super state in case of error/failure or *fail* or *term_recv* message. As soon as all Slave and the Master are in *done* state, the Master will send a termination request, *(*term_req) to all Slaves. In case either of the connected Slaves respond with non-acknowledgement, *term_nack*, all Slaves will be messaged to abort of the termination process (Figure 2.8). On the contrary, if all the connected Slaves acknowledge the *term_req* sent by Master, the Master terminates.

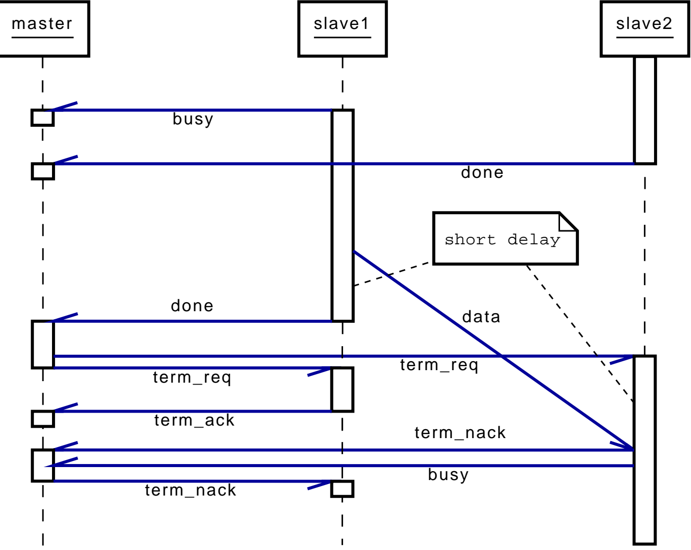


Figure 2.8: SystemC Distributed Library termination request handling.

Using the programming model for SHAPES DOL as in Figure 2.1, the functional simulation using SystemC Distributed Library requires the following steps, resulting in the structure shown in Figure 2.9:

- Parse the XML file to determine the topology of the process network.

- Parse the *Architecture XML* file to determine the Architecture specification of the underlying hardware (*System IP address* and *port number*).

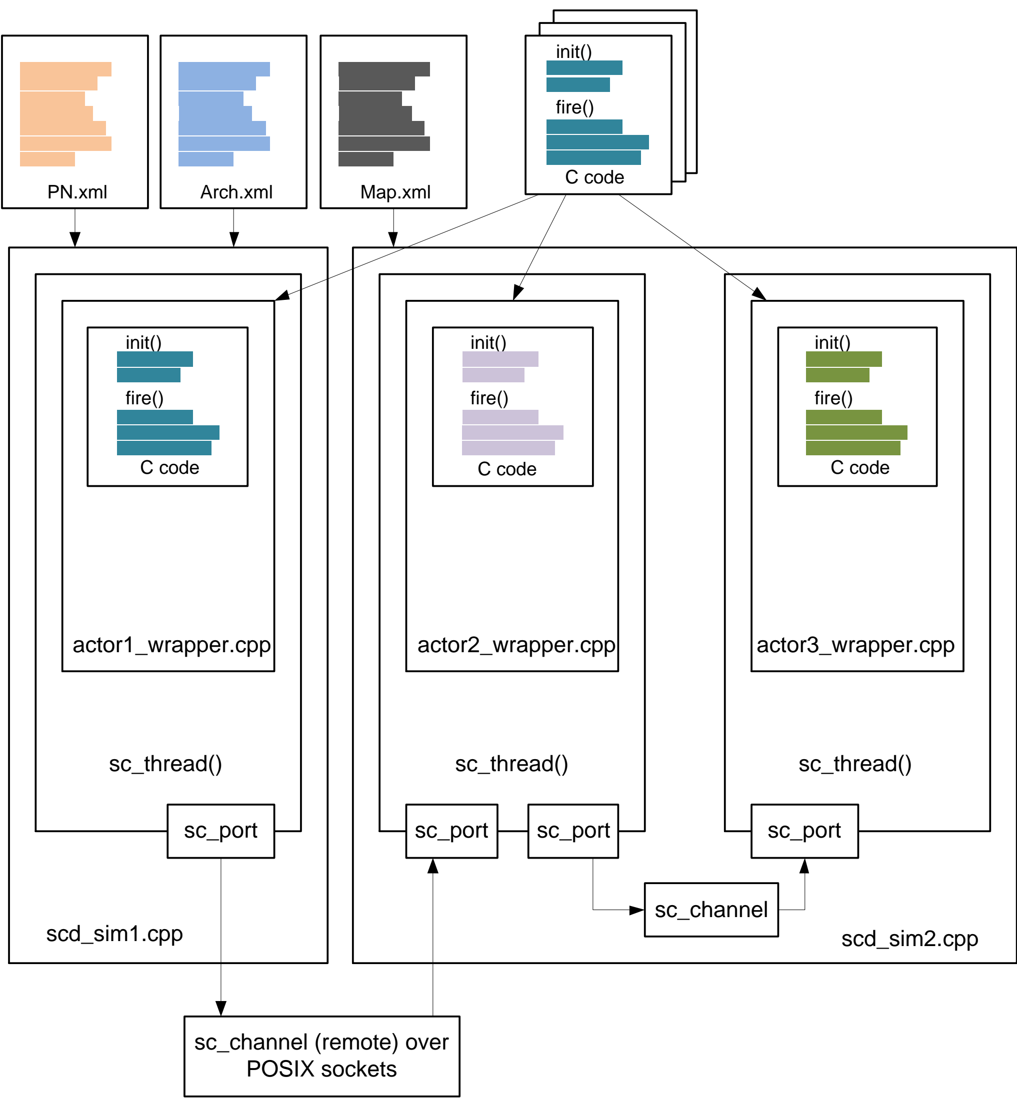


Figure 2.9: Structure of the functional simulation of a Process Network in DOL based on the SystemC Distributed Library for one binary (distributed computational model).

- Parse the *Mapping XML* file to determine the actor/task mapping information.
- Create the directory structure for proper code organization.
- Create a main file for bootstrapping the process network per processor (derived from Architectural specification):
  - Create a *SC_ THREAD* that initializes all the actor/task *init()*.
  - For each *actor/task*, create a *SC_ THREAD* that repeatedly calls the *fire()* routine depending upon the state of the *detach* flag.
  - Instantiate *FIFO channels* (both local and remote) and bind each port in *actor/task* to its corresponding *FIFO channels* (based in SystemC primitive interface and SCD Library).
- Create a *Makefile* to facilitate the build process of the functional simulation for all the SystemC Applications.

- Distribute the binaries based on *IP Address* and execute the binaries in parallel.

Please refer to Appendex A.1 for more information.

2.4 Distributed Application Layer

The Distributed Application Layer (DAL) is a middle-ware design for model-driven development [30] of multiprocessor streaming applications on tiled architecture and a stacked architecture. It is being developed at the Computer Engineering Group at ETH Zürich in the context of a European Framework Programme 7 Research Project called EURETILE [29].

2.4.1 EURETILE

The EUropean REference TILed architecture Experiment (EURETILE) project aims at investigating and implementing a brain-inspired model with a system architecture of massively parallel tiled computer architectures and the corresponding programming paradigm. The target architecture is a fault-tolerant many-tile hardware platform, equipped with a many-tile simulator. A set of software processes (hardware mapped) are generated by the holistic software/middleware tool-chain using a combination of analytic and bio-inspired methods. An elementary tile is a multi-processor, which includes a Distributed Network Processor (for inter-tile communication), a floating-point VLIW processor (for numerical intensive computations), and a RISC processor (for control, user interface and sequential computations). The EURETILE project also investigates and implements the innovations for equipping the existing elementary hardware tile with high-bandwidth, low-latency brain-like inter-tile communication.

2.4.2 EURETILE DAL

The Distributed Application Layer is a middle-ware like the existing SHAPES DOL Middleware with additional features. The goal of the thesis is extend the DOL middleware with DAL middle-ware requirements like:

- Many co-executing and co-existing applications.
- Run-time control of application execution based in system or user-defined events.
- Fault tolerant system.

2.4.3 The Programming Model

The DAL programming model is derived from the Y-API programming interface [40], (can also be termed as Ψ-API programming interface) Figure 2.10, with an extension of the Kahn Process Network at heart for computational model [5], similar to SHAPES programming model Section 2.2.3. The inter-process/task communication primitives remains the same as SHAPES/DOL middle-ware, Table 2.2. A major extension lies in the *mapping* specification. In DAL middle-ware, the mapping file is computed using the specifications mentioned below:

- Process Network Specification.
- Finite State Machine Specification.
- Architectural Specification.

Refer to Chapter 3, Chapter 4 and Chapter 5 for more details information on DAL middle-ware and for visitor specific implementation.

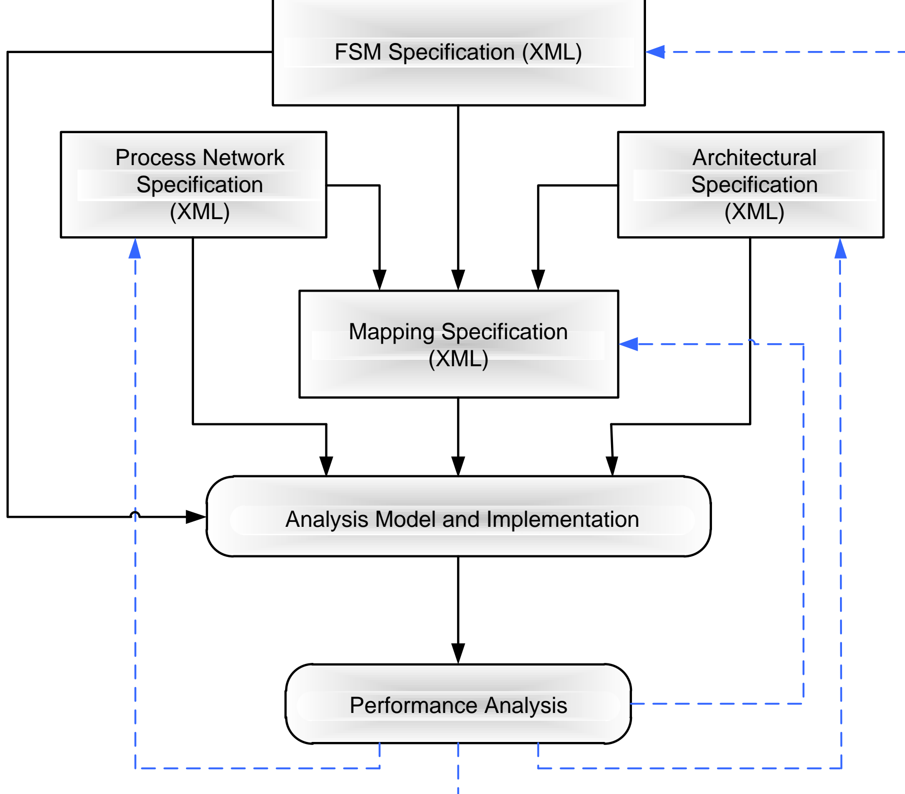


Figure 2.10: Ψ-chart (Modified Y-chart) approach for designing multiprocessors and multiprocessor software for EURETILE platform.

3
Dynamic Systems

3.1 Overview

This chapter contains the key concept involved in the creation and designing the DAL middleware for EURETILE architecture and generic applications structure pertaining to it. The section in this chapter describe the concept of *dynamic system*. This includes the specification of multiple DAL applications, a controller mechanism for managing *dynamic behavior* of co-executing applications. This chapter also describes the *task remapping/migration* feature of DAL middle-ware.

3.2 Problem Definition

The problem definition are listed below:

- Specification of multiple applications.
- Run-time control specification for multiple applications.
- Task migration or remapping based on current *FSM state* of the system.

3.3 Multiple Applications

As described in previous Section, the main goal of this work to define a model that implements a real world scenario of applications. In other words, at any given time frame any given application can be spawned, killed, paused or restarted based on a given user or system event. This is one of the key ideas behind the creation of DOL/DAL middleware for EURETILE [29]. As the time frame specifying the liveliness of an application is not known at the compile time, a low-cost low-overhead control based mechanism has to be devised to handle the application control trigger.

This section deals with the specification of multiple applications. As a given application is composed of multiple processes, each individual process pertaining to a given application is uniquely identified by an application prefix (Figure 3.1). The same constraint is applicable to the software channels, connection labels and input-output ports. A brief illustration of this is shown in Appendix A.2.

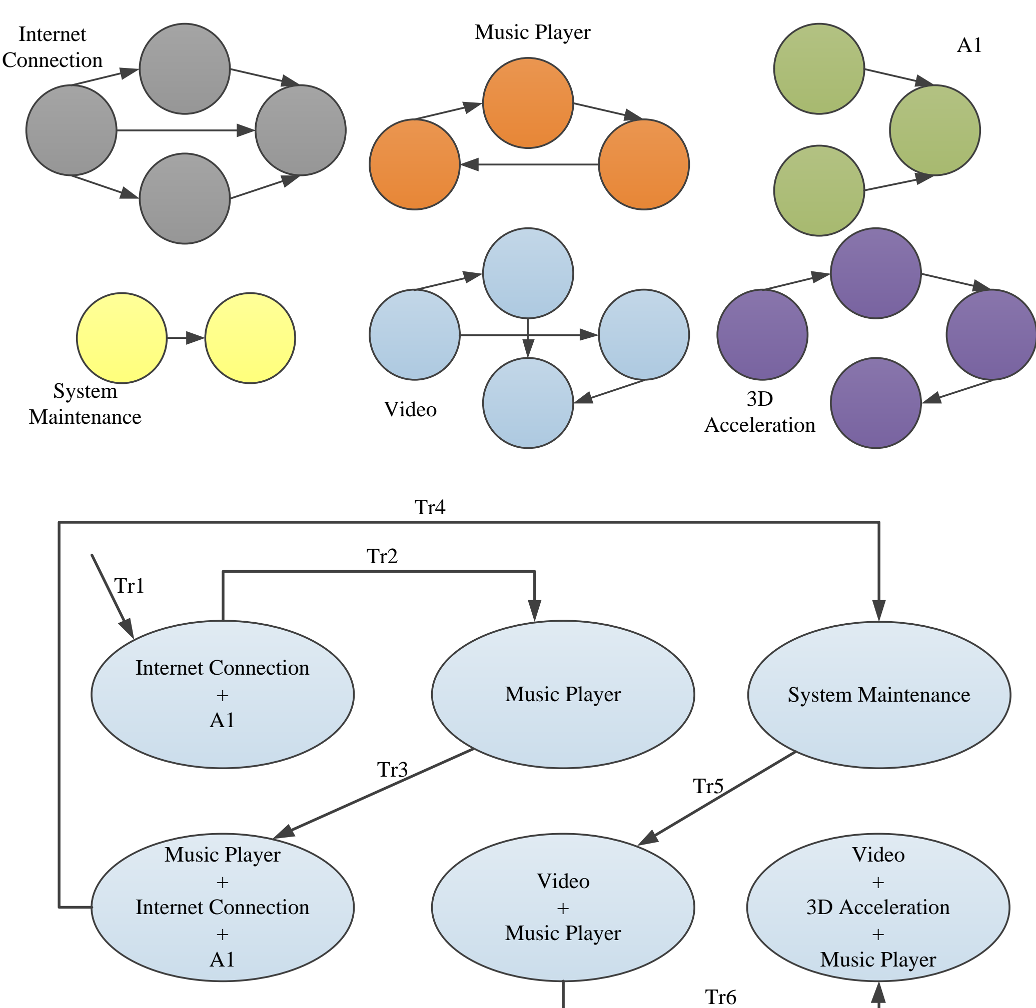


Figure 3.1: Pictorial view of Multiple KPN Applications with FSM (without Start and Finish State) describing Application behavior.

As per DOL software-flow, Figure 3.3, post-parsing the *processNetwork.xml* file, a database containing all application information is created and respectively populated with parsed information. As the input to this design flow is governed by the input *processNetwork.xml*, the user in DAL middle-ware needs to aggregate all the selected application specification into a single *processNetwork.xml* file. The Figure 3.4 shows the software flow followed in DAL middle-ware with an extra step to parse the *FSM.xml* file. The Lisiting A.2.1 shows a simple example of two application specifications in a single *processNetwork.xml* file. The applications shown are two simple model of *producer-actor-consumer*. Each task is uniquely identified, Appendix A.3.

Consider for instance, a DAL user has to design a system with $\mathcal{N}$ number of applications. The *process network* for the entire set-up can be defined using Equation 3.1. The term PN_u signifies the user-defined *processNetwork.xml*.

$$PN_u = \sum_{i=1}^{\mathcal{N}} PN_i \tag{3.1}$$

An application instance can be defined using the element PN_i. The element PN_i can be defined as a tuple of tasks, connections and channels as seen in Equation 3.2.

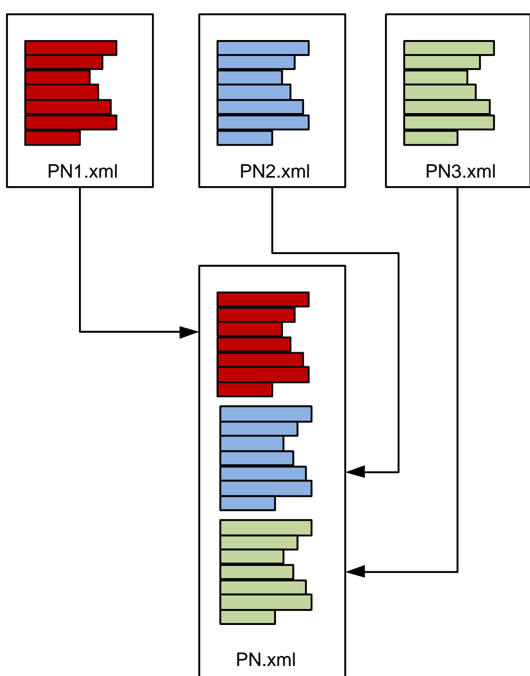


Figure 3.2: Pictorial view of Multiple PN Applications Merged to Single *processNetwork.xml*.

$$PN_i = [T_i, Con_i, Ch_i], \forall i \in \mathcal{N} \tag{3.2}$$

$$T_i = \{T_i^j : \forall i \in \mathcal{N}, \forall j \in \alpha_i\} \tag{3.3}$$

$$Con_i = \{Con_i^j : \forall i \in \mathcal{N}, \forall j \in \beta_i\} \tag{3.4}$$

$$Ch_i = \{Ch_i^j : \forall i \in \mathcal{N}, \forall j \in \gamma_i\} \tag{3.5}$$

The set of *tasks* T_i described by the user-defined application PN_i can be described using the Equation 3.3. Each *task* T_i^j is identified using a *task_id*, a set of input-output *port* and the user generated file containing the specific functions or routines, *source_code* (in *processNetwork.xml* a *task* is termed as a *process*). The element Con_i is an aggregation of *connections* definition between a *channel* and a *task*, Equation 3.4. The third element Ch_i of the tuple PN_i as defined in Equation 3.5 is a collection of *channel* informations. The term α_i defines the number of tasks, β_i is the number of connections and γ_i is the number of software channels defined for a given instance of application PN_i. Each *connection* Con_i^j and *channel* Ch_i^j is defined as a Vector quantity. Each of these elements are defined based the direction of data flow (hence defined using an *origin* and a *target*). All these informations are derived after parsing the *processNetwork.xml* supplied by DAL user. The user defining the file *processNetwork.xml* should uniquely identify each application set using an *App_Prefix* and the each element in the superset of T_i, Con_i and Ch_i should be uniquely identifiable (refer to Appendix A.3 for nomenclature and naming conventions).

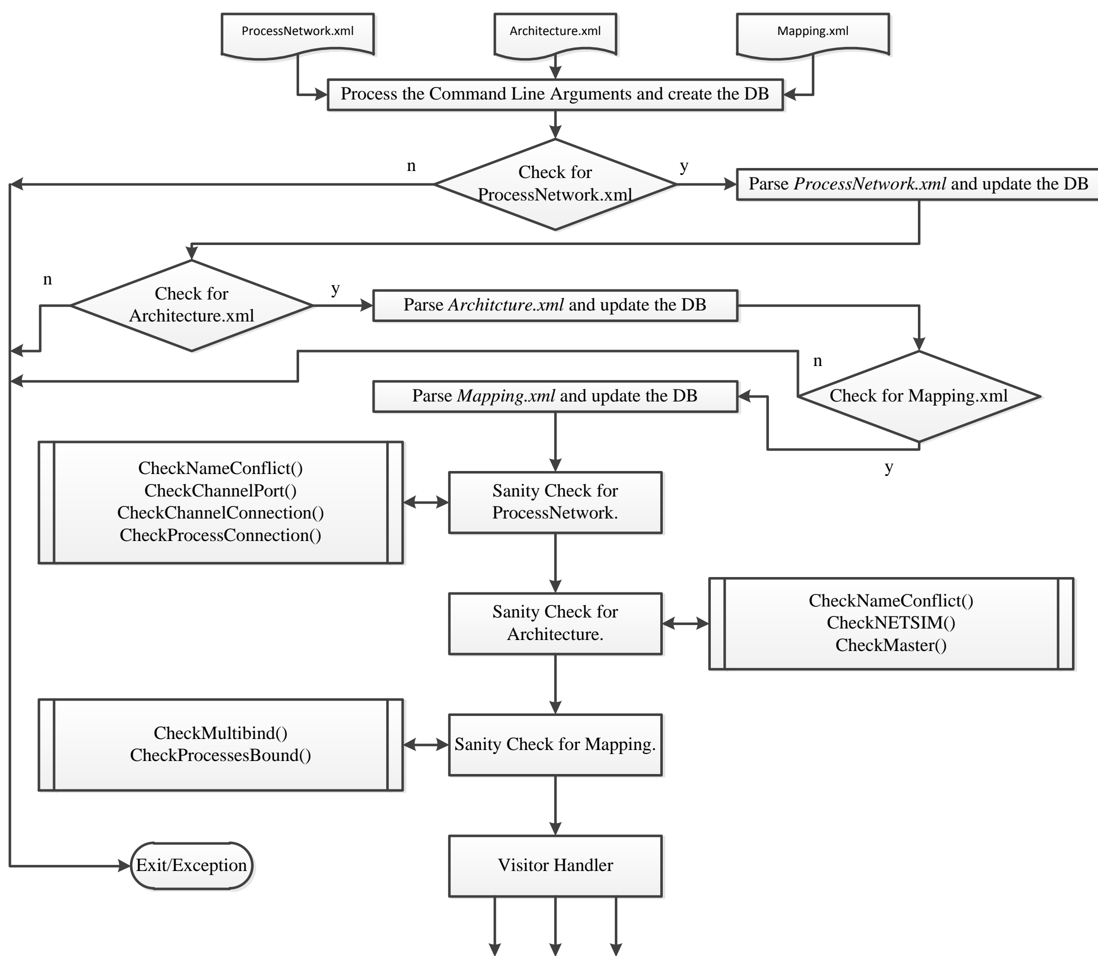


Figure 3.3: DOL data parsing and processing design flow.

3.4 Application Control

As mentioned in this Chapter, the current system should be redesigned to accommodate a substantial number of applications (as opposed to single application on SHAPES DOL platform), a control logic is required to manage the applications PN_i and its respective tasks sets T_i(Equation 3.3). As shown in Figure 3.6, a layer approach is adopted to realize the requirement specification.

The run-time behavior of the DAL applications in general can be termed as *behavioral dynamics*. This idea can be further explained using the following properties:

- Dynamic Control Specification: The DAL user defines and specifies the applications dynamics in the form of a Finite State Machine (FSM).
- Dynamic Application Controller: The layer of DAL middle-ware manages the application based on the Dynamic Control Specification.

3.4.1 Dynamic Control Specification

The application selection for EURETILE platform is decided by the platform user. Along with the *processNetwork.xml* file and the corresponding *source code* files for all individual *tasks*, the

user also specifies a FSM describing the run-time behavior of the applications as a *FSM.xml*. The Figure 3.7 describes a generic FSM that can be provided by the EURETILE/DAL user.

$$FSM = [S_0, S_{final}, State, \rightarrow, Act] \tag{3.6}$$

In general, any given FSM can be described using the tuple in Equation 3.6. The element S_0 denotes the *start state* of the system. S_{final} defines a collection of *final* or *termination states* of the system. All the *state* describing the liveliness of applications including the S_0 and S_{final} is an element of *State* set, Equation 3.7. The $\rightarrow$ defines the list of *transition* and *Act* is a collection of *action*.

$$State = \{State_l : l \in N_{State}, \forall\{S_0, S_{final}\} \in State\} \tag{3.7}$$

$$State_l = [StateId_l, Transition_l] \tag{3.8}$$

The system transition from one state to another is triggered by an event or a collection of events. Hence each state is contains a list of *transitions*, namely $Tranistion_l$ and a unique state identifier, $State_Id_l$ (Equation 3.8).

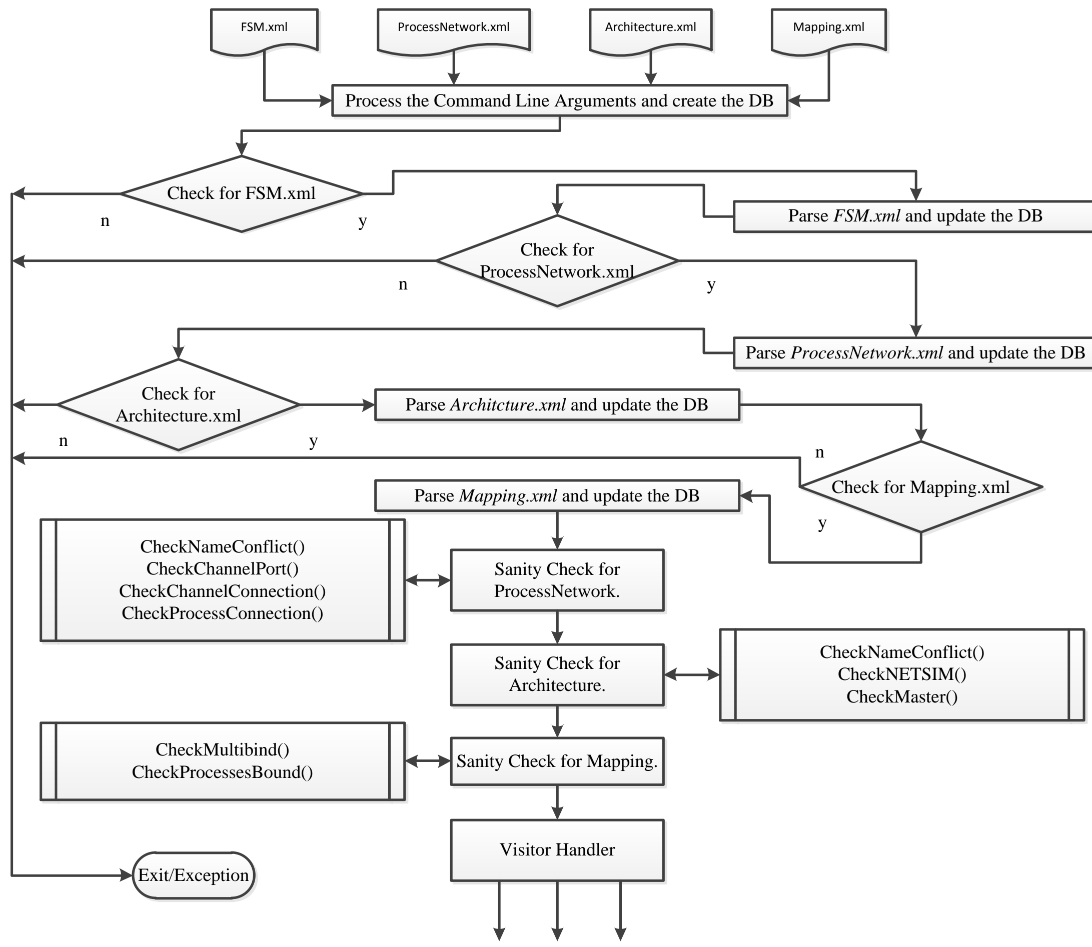


Figure 3.4: DAL data parsing and processing design flow.

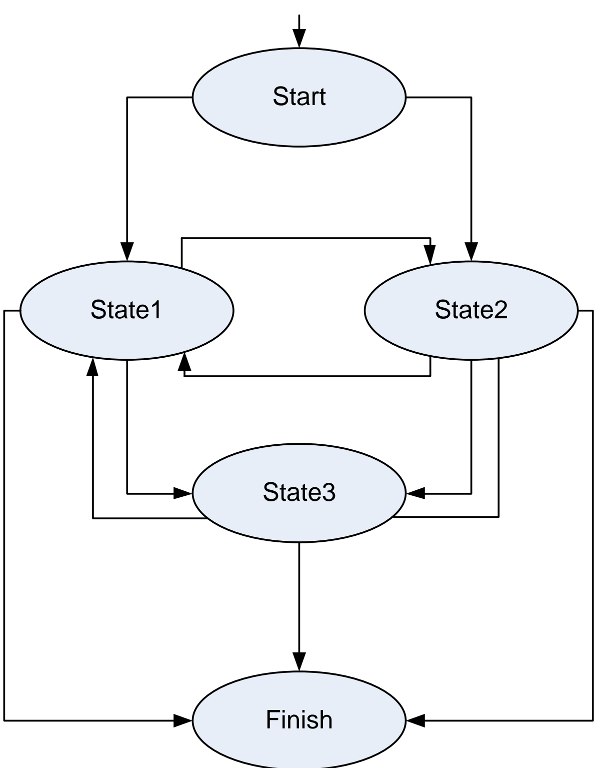


Figure 3.5: A sample FSM for DAL Applications.

$$\begin{aligned} Transition_l = &\{[Event_Id_l, Pr_l, Action_l^m] \\ &: Action_l^m = [App_Prefix_l^m, DAL_Action_l^m] \\ &: \forall m \in \{N_{Transition_l}^{Action}\}\} \end{aligned} \tag{3.9}$$

Table 3.1: Event Description (Figure 3.1)

Event	Description
Ev_1	Active Internet Connection
Ev_2	Activate Music Player
Ev_3	Activate Application A1
Ev_4	Activate System Maintenance
Ev_5	Activate Video
Ev_6	Activate 3D acceleration

A given *transition* $Transition_l$ can be defined by using a tuple of elements namely, an event identifier $Event_Id_l$ or a list of event for trigger , a static priority value Pr_l for ordering the $Tranistion_l$ in a given $State_l$, and a list of action $Action_l^m$ to be taken for successful *transition* (Table 3.1 and Table 3.2). The generic element $Action_l^m$ has been specifically defined to adapt to DAL requirements. An *action* definition consist of an application identifier called as *App_ Prefix* and application activity mode called as *DAL_ Action*. The *DAL_ Action* can be either of the following activity modes:

- START: An application is initialized and started (application is activated, local data (task-specific) initialized).

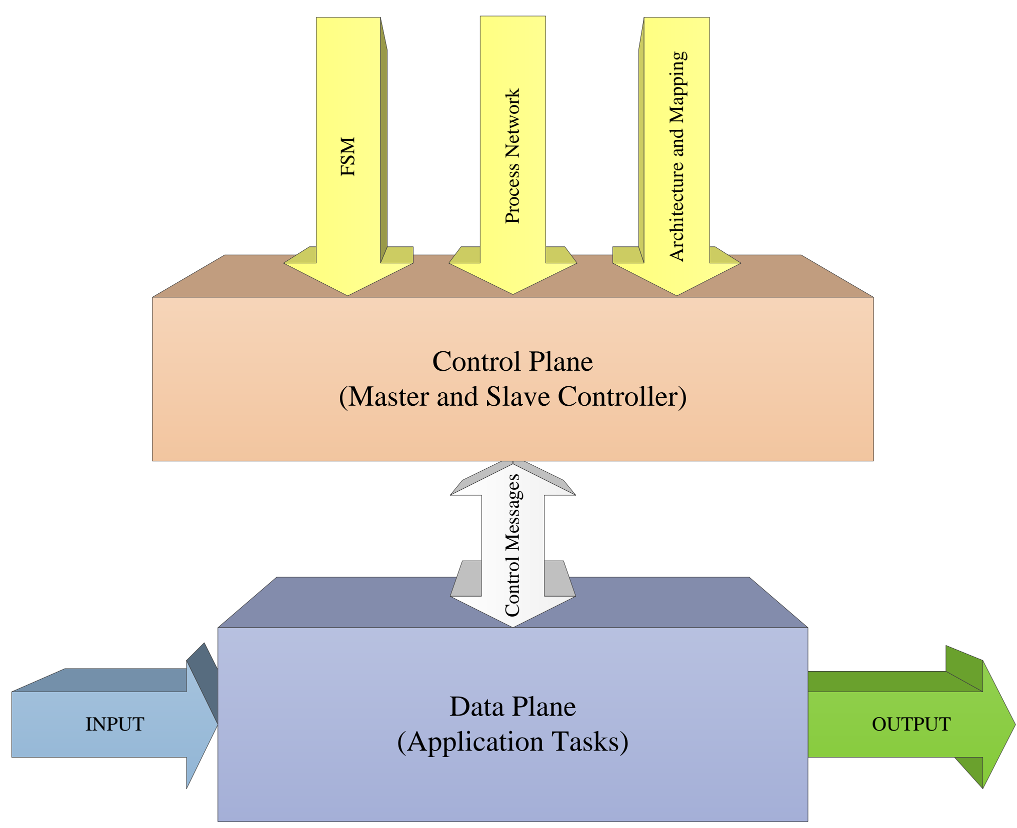


Figure 3.6: Control Plane and Date plane bifurcation of DAL middle-ware.

Table 3.2: Transition-Event Relationship with Actions (Figure 3.1)

Transition	Event	Action
Tr_1	$Ev_1 \parallel Ev_3$	START Internet Connection, START A1
Tr_2	Ev_2	START Music Player, PAUSE A1, PAUSE Internet Connection
Tr_3	$Ev_1 \parallel Ev_3 \parallel Ev_2$	RESUME Internet Connection, RESUME A1
Tr_4	Ev_4	START System Maintenance, STOP Internet Connection, STOP A1, STOP Music Player
Tr_5	$Ev_5 \parallel Ev_2$	START Video, START Music Player
Tr_6	$Ev_6 \parallel Ev_5 \parallel Ev_2$	STOP Video, STOP Music Player START 3D acceleration, START Video, START Music Player

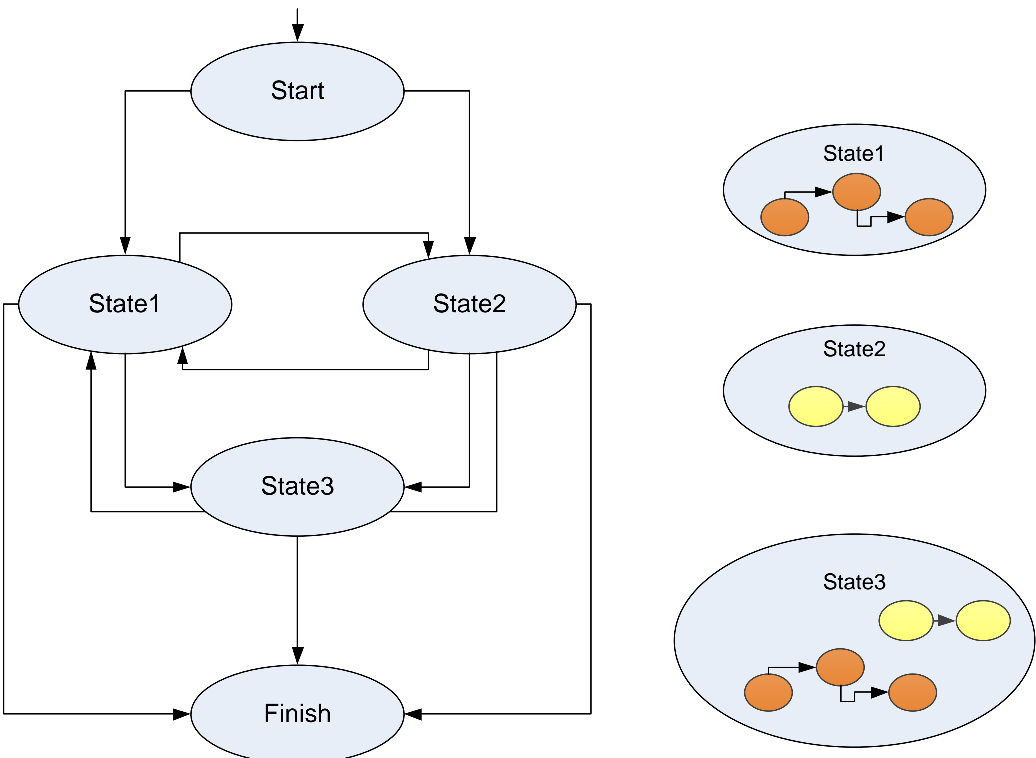


Figure 3.7: FSM for two DAL application with Process Network specification and State-based Application liveliness.

- STOP: An application has finished its tasks and stopped (application is deactivated, local data (task-specific) is cleared).
- PAUSE: An application has paused but is still active (application (task-specific) local data is maintained/preserved).
- RESUME: An application has resumed from the pause phase (application (task-specific) executes with preserved local data).

3.4.2 Dynamic Application Controller

A Control-Plane in Figure 3.6 inputs the Finite State Machine (Section 3.4.1) and implements the dynamic behavior. The Control Mechanism can be implemented in following ways:

- Centralized Control: The Centralized Control paradigm is based on the creation of a intelligent *master controller* on the $Master_{processor}$ and a non-intelligent *slave controller* running on $Slave_{processor}$. The role of the *master controller* is to manage every dynamic decision of the system, namely, detect dynamisms, initiate task migration or fail-over-takeover decision according to the FSM status and processor (μP) status, and dispatch requests to the respective *slave controllers*. The main advantage of the Centralized Control approach is ease of implementation. The *master controller* maintains a system-level database required for dynamic control. It processes *Ev* from *EventList* based on the global system *state* and builds the control message and transmits to respective $Slave_{processor}$. The *slave controller* receives the control message and calls *DAL_Action* for the listed *tasks* in the control message. This enable a simple logic at the *slave controller* level of the control hierarchy. On the downside, the *master controller* is the root of all control decision and depending on size

and complexity of input specification (process network, FSM, Architecture and mapping), the algorithmic complexity (with respect to space and time) of dynamism-handling routine increase considerably.

- Distributed Control: The Distributed Control approach is built on the hypothesis that each processor in the system is autonomous and shares global system-level management information. There is still a existence of non-intelligent *master controller* with an intelligent *slave controller*. The *master controller* collects system-wide events and broadcasts events and control messages (send from a *slave controller*) to all the *slave controllers*. Each *slave controller* manage and maintains dynamism related decision data and control at local processor level. The *slave controllers* need to synchronize the dynamism-related database amongst themselves to seamlessly handle the events and take a coherent control decisions like task migration or fail-over-takeover decision according to the FSM status and processor status. The main advantage of this model is that all the *slave controllers* exhibits similar behavior, hence advantageous at code-generation level. It also ensures ease of management when the architecture scales to higher degree. On the contrary, there are several disadvantages to this policy. The bus connectivity would be a major bottle-neck due to broadcast messages from *master controller* and synchronization of databases across *slave controllers* for data coherence.

- Hybrid Control: The proper design of trade-off adaption from both Centralized and Distributed Control approach can lead to Hybrid Control system. This paradigm relies on the brain-inspired hierarchical control, namely clusters of processors Figure 3.8. The Hybrid Control uses the concept of Centralized Control at the First Level of hierarchy i.e. at inter-cluster control and the Distributed Control within a cluster i.e. for intra-cluster control. This Control model has most of the advantages of both the both the Centralized and Distributed Control with relatively minor drawbacks.

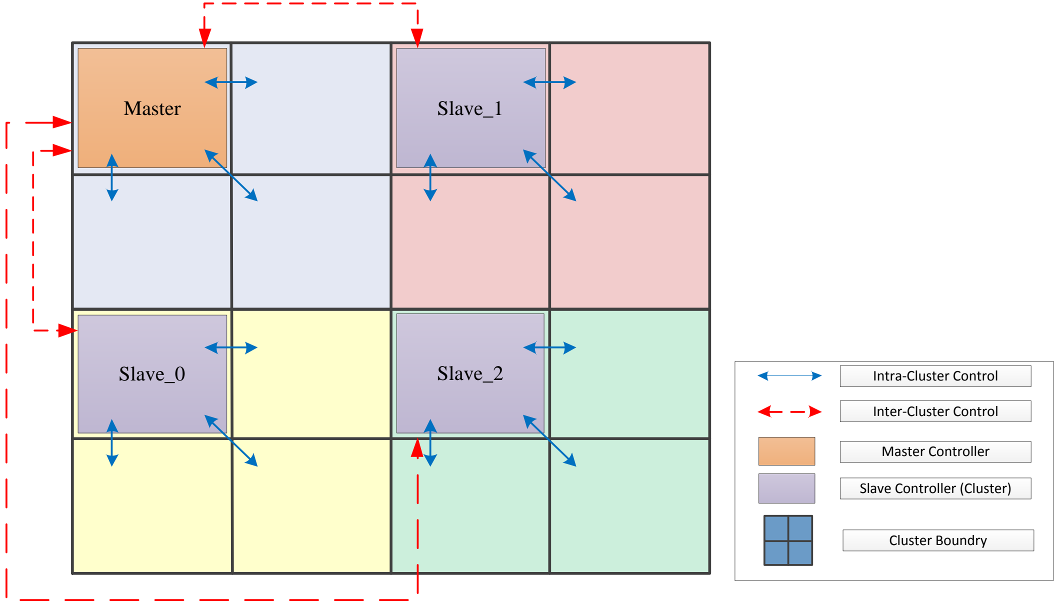


Figure 3.8: Control Hierarchy for Cluster MPSoC.

The adaption of either Centralized or Hybrid Control would be justifiable in DAL context. If the architecture scale to higher magnitude (Table 2.1), it would be logical to migrate to Hybrid Control instead of Centralized control for faster system response time and ease of management.

In either of the Centralized or Hybrid Control model implementation, the *master controller* and the *slave controllers* have to be implemented as execution threads or as processes. Hence a

CtrlprocessNetwok.xml has to be defined depending on the architecture of the system (Appendix A.2). Mathematically, the *controller process network* can be defined as Equation 3.10, where $Ctrl_PN^{Master}$ describes the *master controller process network*, a generic *slave controller process network* is described as $Ctrl_PN_j^{Slave}$ and $\mathcal{J}$ is the number of $Slave_{processor}$.

$$PN_{ctrl} = \sum_{j=1}^{\mathcal{J}} Ctrl_PN_j^{Slave} + Ctrl_PN^{Master} \tag{3.10}$$

$$Ctrl_PN^{Master} = [Ctrl_T^{Master}, Ctrl_Con^{Master}, Ctrl_Ch^{Master}] \tag{3.11}$$

$$\begin{aligned} Ctrl_PN_j^{Slave} =& [Ctrl_T_j^{Slave}, Ctrl_Con_j^{Slave}, Ctrl_Ch_j^{Slave}] \\ & \forall j \in \mathcal{J} \end{aligned} \tag{3.12}$$

$$Ev_Id_\lambda \in EventList, \forall \lambda \in N_{EventList} \tag{3.13}$$

$$\begin{aligned} \mathcal{TS} =& \{[Ev_Id_\alpha, S_\beta^C, S_\beta^N, S_0, S_{final}] \\ & : \forall \{S_\beta^C, S_\beta^N, S_0, S_{final}\} \in State, \\ & \forall Ev_Id_\alpha \in EventList, \forall \alpha \in N_{Event_Id}, \\ & \forall \beta \in N_{State}\} \end{aligned} \tag{3.14}$$

The event identifier Ev_Id_λ triggering the *state-transition* from one *State* to another is managed by the *master controller task* in either Centralized or Hybrid Control Model. The current list of *event* identifier is maintained in the *EventList*, Equation 3.13. The *Transition System* $\mathcal{TS}$ is the methodology followed by the *master controller* or/and *slave controller* for making a successful *state-transition*. The $\mathcal{TS}$ is defined in Equation 3.14 with elements as *event identifier* Ev_Id_α, *current state* and *next state* of the system (S_β^C, S_β^N) and the *start state* and *finish state* of the system (S_0, S_{final}).

3.5 Task Remapping/Migration

The Figure 3.10 depicts the run-time behavioral dynamisms of the *DAL application* using the user-defined FSM. Using statistical performance analysis methods and DSE a series of *FSM state-dependent optimal mapping* strategies can be proposed. In other words, for efficient resource utilization within constraints and system bounds, each intermediate $State_\alpha$ of the FSM ($\forall State_\alpha \notin \{S_0, S_{final}\}$) can be described with a distinct *mapping* strategy. Consider the *behavioral dynamics* of the system in-terms of *liveliness* of *applications* based in controller messages, Figure 3.9.

Each application *liveliness node* can be uniquely described by a set of states with an *optimal mapping strategies/solutions* that can vary from one *FSM state* to another or from one operation mode to another (Figure 3.9 one optimal mapping strategy).

Consider for instance, a *static optimal mapping* for an application. It can be described as uniform distribution based on number of *tasks* on given *processors* (here consider $\mathcal{J} = 3$; in general $\mathcal{K} \ll \mathcal{N}$), Table 3.3. This *application task* distribution requires existence of some selected *tasks* (consider $\mathcal{K}$ *task* have multiple existence) on more than more $processor_\beta$ ($\forall \beta \in \mathcal{J}$). As each *application task* should be uniquely identifiable (Appendix A.3), the replication of *tasks* has to be addressed with a different *task identifier* $\overline{T}_k$, also termed as *migrated task*; this feature is

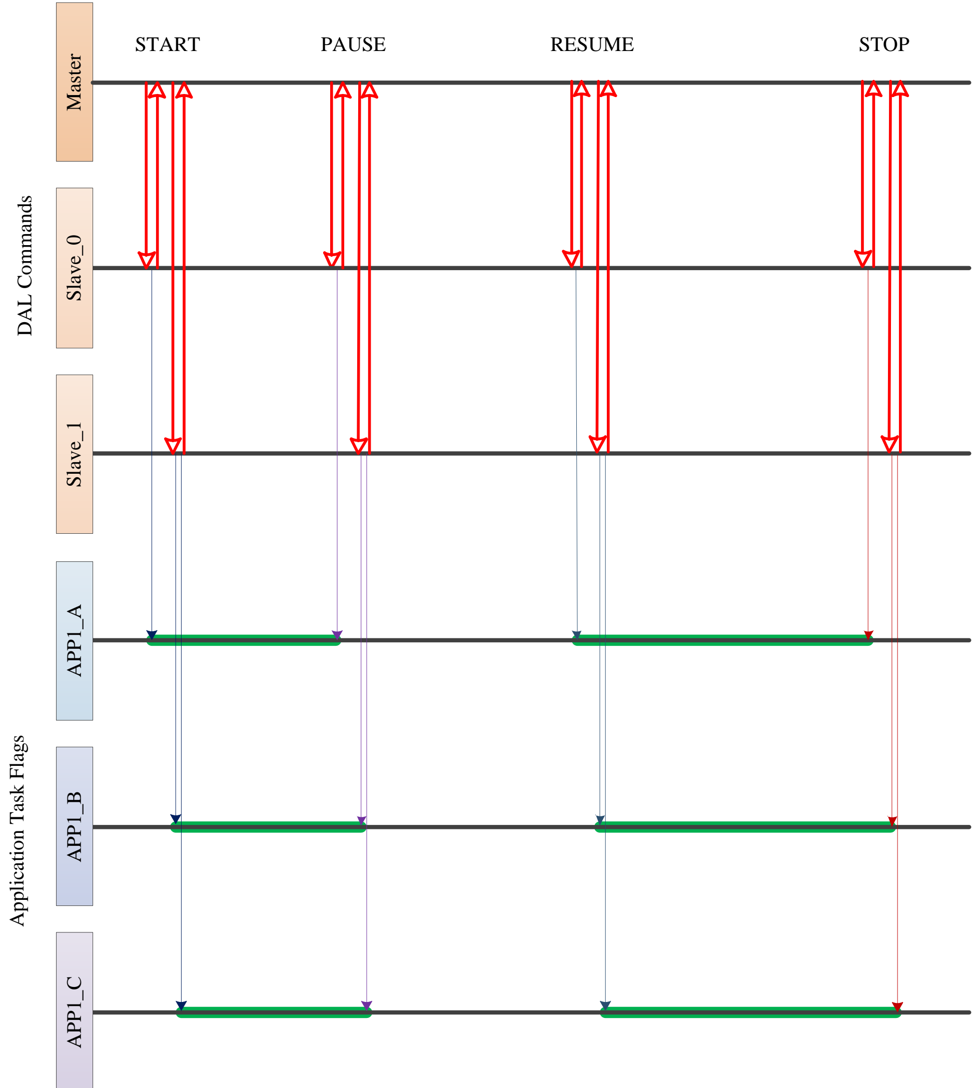


Figure 3.9: Behavioral Dynamics for Single Application distributed over two Processors.

Table 3.3: Application Task Mapping Strategies ($\mathcal{J} = 3/\mathcal{K} = 12$).

Mapping Strategies	Task Distribution
Optimal	$[4, 4, 4]$
$State_{\alpha}$	$[3, 3, 6]$
Temperature Aware	$[7, 1, 4]$

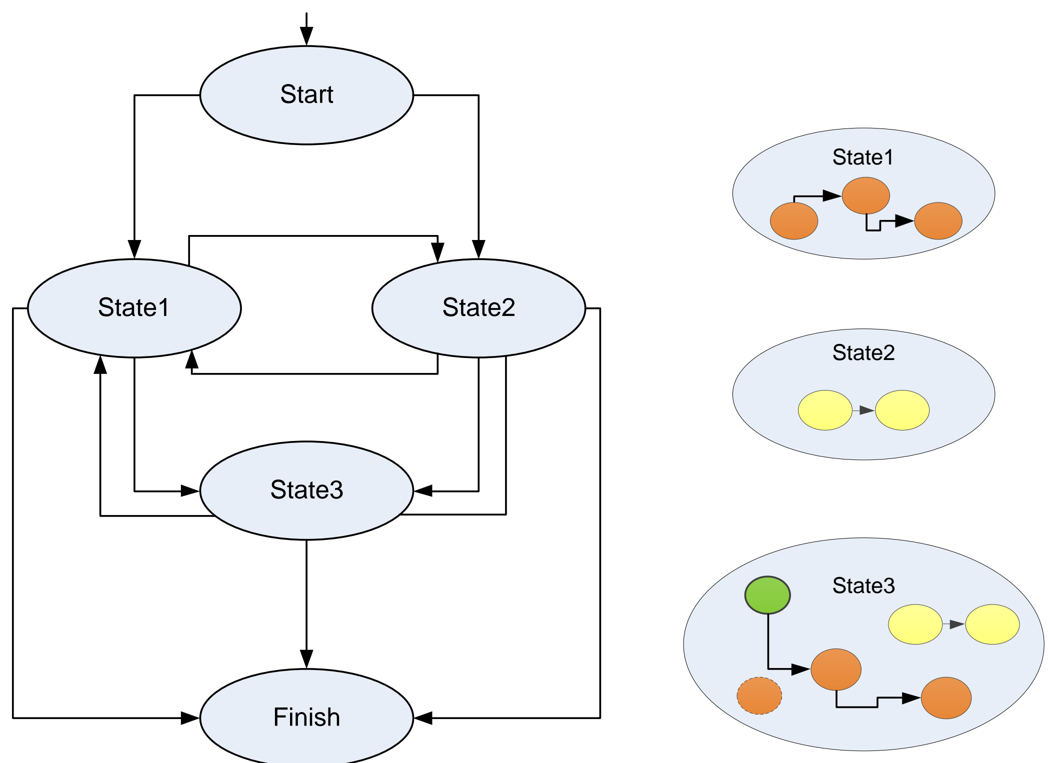


Figure 3.10: Finite State Machine for Applications ($N = 2$) and State specific Optimal Mapping.

called as *task migration/remapping*. Similar methodology is adopted for defining the *migrated connections*, $\overline{Con}_k$ and the *migrated channels*, $\overline{Ch}_k$ ($\forall k \in \mathcal{K}$).

Traditionally, static mapping for a single application (DOL scope), mapping optimization was based on objectives like *timing-behavior*, *hardware cost*, *bandwidth*, or *load balancing*. But in dynamic system of DAL, the *reconfiguration/migration cost* is described in terms of *migration time* and *migrated resource utilization* has to be considered. These decisions can be computed both statically, using performance analysis and DSE, and dynamically at run-time, based on current system utilization. An ideal methodology would be to compute this configuration statically, since the number of *applications* $\mathcal{N}$ is statically known along with its *liveliness* information for any given system *State*. On the upside, the *statically* computed *task migration* happens in a seamless manner without any run-time system analysis overhead as against *run-time task migration* decision making. On the downside, *run-time task migration reconfiguration* decision is not possible in the *statics* approach.

$$T = \{T_i \cup \overline{T}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\alpha}_{\mathcal{K}}\} \tag{3.15}$$

$$Con = \{Con_i \cup \overline{Con}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\beta}_{\mathcal{K}}\} \tag{3.16}$$

$$Ch = \{Ch_i \cup \overline{Ch}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\gamma}_{\mathcal{K}}\} \tag{3.17}$$

In nutshell, the elements $\overline{T}_k$, $\overline{Con}_k$ and $\overline{Ch}_k$ describe the set of *migrated tasks*, *migrated connections* and *migrated channel* respectively derived from it original user-defined specifications ($\forall k \in \overline{\alpha}_{\mathcal{K}}$, $\overline{\beta}_{\mathcal{K}}$ and $\overline{\gamma}_{\mathcal{K}}$ respectively). The aggregate collection of *application tasks* with *State-based task migration*, T, can be described as a super set of *original tasks* T_i and *migrated tasks*

$\overline{T}_k$; using Equation 3.15. In a similar manner, the entire collection of *application connections*, Con, is a super set of *original connections* Con_i and *migrated connection* $\overline{Con}_k$; using Equation 3.16 and the *Software channels*, Ch is a super set of *original channels* Ch_i and *migrated channel* $\overline{Ch}_k$; using Equation 3.16.

3.6 Implementation

This section deals with the implementation aspect of building up the the DAL database with correct information for visitor based code generation.

3.6.1 Multiple Applications

Multiple applications are defined in an application scenario. Each application is defined as per the nomenclature A.3. A preprocessing module can be created to generate the file to suit this nomenclature. In the current implementation, the generation of a process network file (in xml format) is constructed by hand.

3.6.2 Finite State Machine

The DAL middle-ware is built on top of DOL framwork, hence the requirement for application control based on FSM needs to be integrated to the model. A new simple FSM schema (*fsm.xsd*) is designed for reading and processing the dynamic application behavior model.

Algorithm 3.1 FSM.xml File Parsing.

Require: SAX Parser Library; Stack Library
Ensure: $fsm_path \neq NULL$ and fsm_path is valid
  Create Stack element $_stack$
  Create XML Parsing element $_xml2fsm$
  Clear $_stack$
  Parse (fsm_path) **and Process Entities**
  $_fsm \leftarrow$ **Pop** $(_stack)$

A parse is required for parsing the XML file and store the FSM entity (Figure 3.11) into the DAL database . The Algorithm 3.1 describes in detail the mode of parsing and storing the information into the DAL database. The parsing is build on SAX parser [49] and JDOM [50] libraries provided for Java Development Environment. (The FSM XML parsing algorithm was adaption from model in [51, 52]).

The Algorithm 3.2 (for storing the details of root xml schema entity - **FSM**) and 3.3 (for parsing the *State* information) is called whenever the parsing Algorithm 3.1 encounters an element **FSM** and **State** in the *FSM.xml*. The Algorithm 3.3 creates a new data structure for element *State* and updates the respective data elements. Once the XML parser encounters the end tag of the element *State*, it pops the content to stack and the added *State* data structure to the FSM *StateList*. Similar approach is followed for parsing and storing the entries of other FSM related elements namely *Transition* and *Action*.

3.6.3 Mapping Modifications

As discussed in above section, a given *application task* can be mapped to multiple processors. This is referred to as *migrated tasks*. To support this feature, the existing DOL mapping schema has to be modified. The *binding* information containing the *processName/taskName* and *processorName*

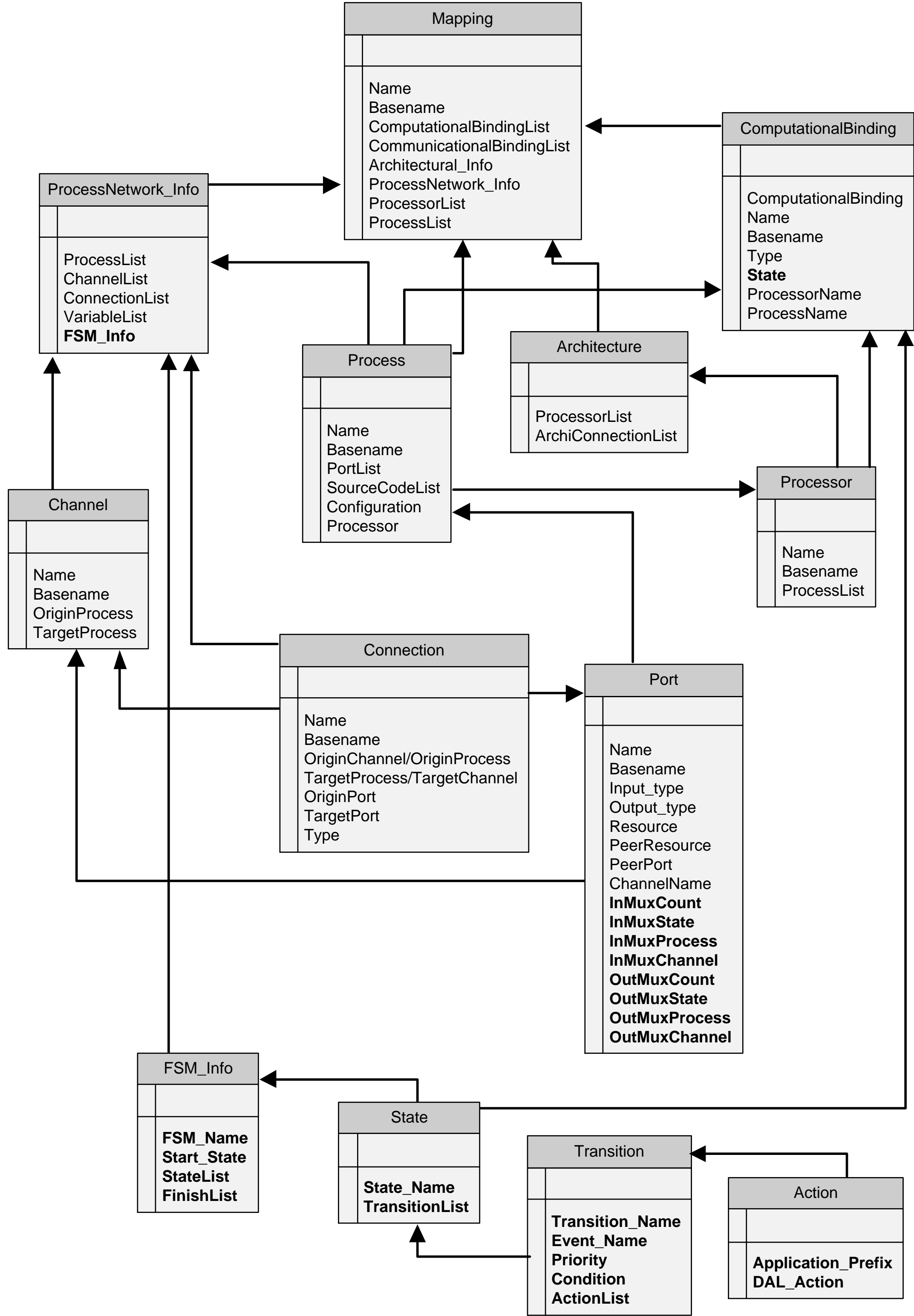


Figure 3.11: DAL entity relationship model.

has to be coupled with *active FSM state* or a list of *active FSM states* (*ComputationalBinding* entity in Figure 3.11). For a DAL application, it is mandatory to specify the content of the sub-entity *State* in *binding* element. In case of controller tasks, the above constraint is not valid. Refer to Listing 3.1 for a sample example with multiple *Active FSM states* for a given *DAL application - APP1*. A software routine 3.4 is added it original *Map Parser* to collect the *state* information and append to the *binding* instance of all *ComputationalBinding* entities.

```
1      <!-- APP1 -->
2      <binding name="APP1_generator" xsi:type="computation" state="
           State1_State4_State6_State9_State10">
3          <process name="APP1_generator" />
4          <processor name="sim1" />
5      </binding>
6      <binding name="APP1_consumer" xsi:type="computation" state="
           State1_State4_State6_State9_State10">
7          <process name="APP1_consumer" />
8          <processor name="sim2" />
9      </binding>
10     <binding name="APP1_square" xsi:type="computation" state="
           State1_State4_State6_State9_State10">
11         <process name="APP1_square" />
12         <processor name="sim2" />
13     </binding>
14     </mapping>
```

Listing 3.1: Mapping information for Application Tasks with multiple Active States.

Algorithm 3.2 Process root FSM information.

procedure $ProcessFSMInfo(attributes)$
▷ Begin processing the FSM Tag
$name \leftarrow attribute.getValue(\mathbf{name})$ ▷ Extract *FSM* sub-entities
$description \leftarrow attribute.getValue(\mathbf{description})$
$startstate \leftarrow attribute.getValue(\mathbf{startstate})$
Create $_fsm(name)$ ▷ Store *FSM* sub-entities
$fsm.Description \leftarrow description$
$fsm.Startstates \leftarrow startstate$
Return $_fsm$ ▷ updated FSM data structure
end procedure

procedure $ProcessFSMInfo(_stack)$
▷ End processing the FSM tag
Pop $(_stack)$
end procedure

Algorithm 3.3 Process FSM State information

Ensure: $attributes \neq NULL$
procedure $ProcessStateInfo(attributes)$
▷ Start of State Tag
$name \leftarrow attribute.getValue(\mathbf{name})$
if $name \neq NULL$ **then**
Create $state(name)$
end if
end procedure

procedure $processStateInfo(_stack)$
▷ End of State tag
$state_tmp \leftarrow \mathbf{Pop}(_stack)$
$fsm \leftarrow \mathbf{Peek}(_stack)$
Add $state_tmp$ to fsm
end procedure

Algorithm 3.4 Update State information for Binding Element

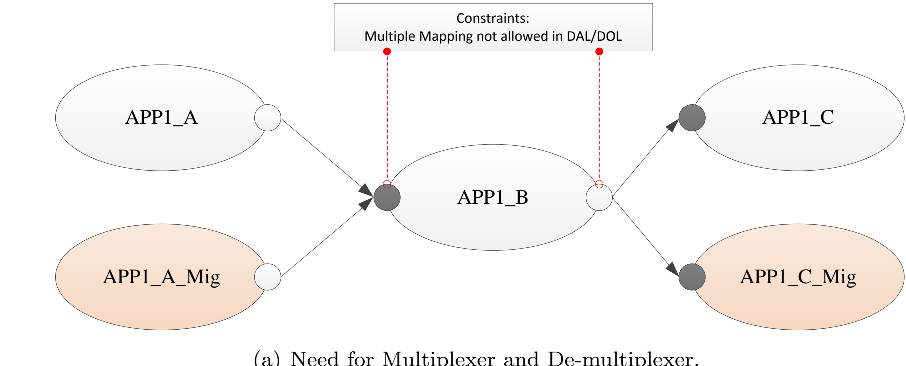
procedure $UpdateStateInfo(attributes)$
▷ Begin processing the *state* Tag
$state_name \leftarrow attribute.getValue(\textbf{state})$
▷ Extract *state* sub-entity from *mapping.xml*
$binding.state \leftarrow state_name$
▷ Store *state* sub-entity
end procedure

3.6.4 Support for Multiple Input/Output Channel Port interfacing in SystemC

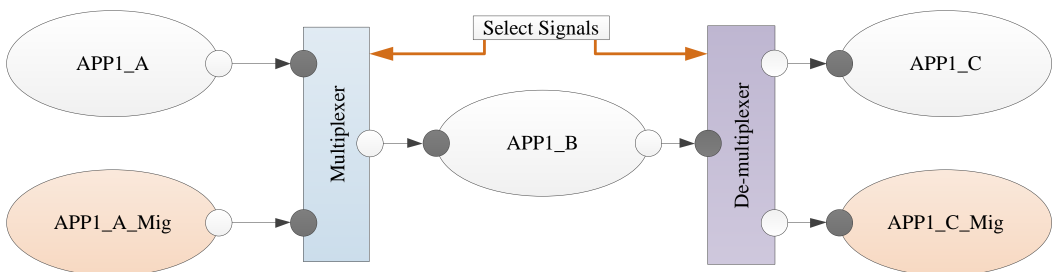
One of the design properties of SystemC standard [44] is the static nature of programming. The SystemC application designer should know the connection information about the *sc_module* and *sc_interface* at design time. Hence the *ports* defined in the *sc_module* and the *ports* defined in the *sc_interface* has to be statically mapped.

This design constraint need to be taken into account while designing a source *sc_module* that is connected to multiple destination (SIMO type) *sc_module* via different *sc_interface*, but at any given time frame the data transmission is active on a single *sc_interface* only; controlled by some signal. The Figure 3.12 shows the pictorial model of this behavior. This model is required to support the two basic DAL requirements:

- Run-time FSM state-based optimal mapping modes using task migration.
- Run-time Fault Tolerance Support using Task Cloning 4.2.



(a) Need for Multiplexer and De-multiplexer.



(b) Placement of Multiplexer and De-multiplexer.

Figure 3.12: Addition of Multiplexer and De-multiplexer.

A simple and scalable method is to attach software blocks called as *multiplexers* at the *input port*

and *de-multiplexers* at the *output port* of the *sc_module*. The Listing 3.2 shows the structure of *multiplexer* and a *de-multiplexer*.

```
struct _Mux_in_APP1_<actor>
{
  sc_port<read_if> MUX_INPORT_State1_State4;
  sc_port<read_if> MUX_INPORT_State6_State10;
  sc_port<write_if> MUX_OUTPORT;
  char select_sig_mux[40];
};

struct _Demux_out_APP1_<actor>
{
  sc_port<read_if> DEMUX_INPORT;
  sc_port<write_if> DEMUX_OUTPORT_State1_State4;
  sc_port<write_if> DEMUX_OUTPORT_State6_State10;
  char select_sig_demux[40];
};
```

Listing 3.2: Multiplexer and De-multiplexers of in and out ports for APP1_<actor>

Addition of *multiplexer* and *de-multiplexer* at *port* enables the use of multiple *sc_port* per *sc_module*. This also ensures that the KPN blocks like the *split tasks* and the *merge tasks* can be easily adapted to run-time DAL requirements. The limitation of this model is that all the instance of a *task/process* have either a *multiplexer* or a *de-multiplexer* or both should behave in a similar way. This clause should be considered during the performance analysis and DSE for a given process network along with FSM and architecture.

Each instance of *multiplexer* and *de-multiplexer* has a select signal namely *select_sig_mux* and *select_sig_demux*, that is used by the *process/task* at run-time to decide which *port* to be select for *DOL_read()* or *DOL_write()*. At run-time the *master controller* sends relevant information about the application-task-port specific *Mux select* or *Demux select* signal (for *Centralized Control/Hybrid Control* model only). In this work, *Centralized Control Model* is adopted.

3.6.5 Database Update for Remapped/Migrated Processes

This section provides the details in terms of algorithmic approach adopted for updating the DAL/DOL database for *code generation* process for a given *visitor*. The process of database update is governed by the addition of *migrated tasks*. The information about *migrated task/process* is added to the DAL/DOL database in two stages.

- In the first phase, the binding information is obtained from the *_map* data structure. The *process.name* information in the bindingList is searched for information like **Migrate**[*]. In Algorithm 3.5, the first procedure ($ObtainNewTaskInfo()$) is used for obtaining a list of *processName* from the *ComputationalBindingList*. Whenever a *processName* satisfies the previous condition (with keyword **Migrate**) and the *processName* matches closest to any *process.name* in the *processNetwork* (original) (Figure 3.11), a new local copy (non-linked) of the *process* instance (e.g. *tmp_Process*) is created and the *name* entity is modified with *processName*.

$$tmp_Process.name \leftarrow processName$$

  The process instance *tmp_Process* is added to *processNetwork.ProcessList* if the entity is not present in the list (using the $UpdateGlobalProcessList()$ in Algorithm 3.5). For

[*] The search methodology for finding the a *sample_text* as a substring in *main_text* is by **parsing** and **comparing** the *main_text* for *sample_text*. This process is used in the Algorithms (3.5, 3.7, etc) and the expression used is:

$$sample_text \ni main_text$$

The functions present in C/C++ and Java library for similar functionality are *strstr()* and *String.contains(substring)*, respectively.

Algorithm 3.5 Updating Computational Binding Information for *Migrated task/process*

```
procedure ObtainNewTaskInfo(_map)
                                        ▷ Obtain Task from the Computational Binding List
    local_pList ← _map.pn.processList                          ▷ A local copy
    local_cBindList ← _map.compBindList
    Create append_pList
    i ← 1
    j ← 1
    repeat
        t_bPName ← local_cBindList[i].Name
        if (t_bPName ∋ Migrate) then
            repeat
                _PName ← local_pList[j].Name
                if _PName ∋ t_bPName then
                    if _PName ≠ t_bPName then
                        Create new_p = local_pList[j]
                        new_p.Name = t_bPName
                        Append new_p to appened_pList
                    end if
                end if
                increment j
            until j ⩽ T_N                              ▷ T_N = local_pList.length
        end if
        increment i
    until i ⩽ B_N                                  ▷ B_N = local_cBindList.length
end procedure

procedure UpdateGlobalProcessList(append_pList)
                                            ▷ Append Processes to Global List
    _procList ← _map.pn.processList                        ▷ Local copy for Update
    i ← 1
    repeat
        t_Proc ← append_pList[i]
        if t_Proc ∉ _procList then
            Append t_Proc to _procList
        end if
        increment i
    until i ⩽ N_L                                 ▷ N_L = append_procList.length
    _map.pn.processList ← _procList                                  ▷ Updated
end procedure
```

efficiency, the addition of new elements to the *ProcessList* is done just once after all the *tmp_Process* have been added to a vector of processes.

- In the second phase of *process* update, the structural elements like the *port* information is updated. This update is done when the new *connection* and *channel* information are created and added to the *ConnectionList* and *ChannelList* respectively. In the following Sections, the procedural call to *UpdateProcessContents*() (Algorithm 3.6) is made whenever a new *connection* is added to *ConnectionList* and *ChannelList*.

Algorithm 3.6 Updating the elements in new *Process* instance

Require: A $_processIns$ instance to be updated
procedure $UpdateProcessContents(_processor, _portList)$
▷ Update Process sub-entities
$_processIns.Processor \leftarrow _processor$
$_processIns.portList \leftarrow _portList$
end procedure

Table 3.4: Migrate Connection and Channel Types (Example)

Type	Origin Process (Origin Connection)	Channel	Target Process (Target Connection)
1	$A/A_Migrate$ $(AC1_Migrate)$	$C1_Migrate$	$B_Migrate/B$ $(C1B_Migrate)$

3.6.6 Database Update for Remapped/Migrated Connections

This section provides the algorithmic approach for creating and updating the DAL/DOL database with additional *migrated connections*.

The interface information of the *process port* to a *channel port* is stored in data structure called *connection*. The instances of connection is broadly classified as:

- Origin Connection: The *origin* resource is a *process* and *target* resource is a *channel*. Hence at database level, the following information is stored:
  - $connection_ins.origin_resource.type \leftarrow NULL$
  - $connection_ins.target_resource.type \leftarrow FIFO$
- Target Connection: The *origin* resource is a *channel* and *target* resource is a *process*. The following information stored is at the connection instance level in the database.
  - $connection_ins.origin_resource.type \leftarrow FIFO$
  - $connection_ins.target_resource.type \leftarrow NULL$

The Table 3.4, shows the sample connections to be created and appended to the *Connection-List* in case of FSM state-based *process/task migration*. The Algorithm 3.7 is used to create a list/vector of *migrated connections* required for connecting the *migrated process port* and *migrated channel port* and vice-versa. A list/vector of unconnected processes is obtained from the DOL function *_checkProcessConnection(_map)*. This list is termed as $new_ResourceList$ (the process entity is inherited from resource entity). The contents of $new_ResourceList$ containing **Migrate** is serially compared (for existence of sub-string) to either origin or target resource name (*Process.name*) of the existing connection instances in the *ConnectionList* to entries in $new_ResourceList$. Upon success, a local copy (non-linked), l_con, of connection instance $(_conList[j] : \exists\, j \in processNetwork.ConnectionList.length())$ is created. The element modification and update at resource level is dependent on the *connection type*. The elements that need to be updated include the origin and target resource (here either process or channel). If the process is to be updated, the PeerResource of the appropriate port is also updated to *l_con*. If the channel entity is to be updated, the origin process and target process are accordingly updated. At channel-port level, the PeerResource pertaining to migrated connection (closest resemblance in name entity) is updated to *l_con*. Post resource updates, the l_con is added to the vector of new connections, *append_conList*. The procedural call *UpdateGlobalConnectionList()* (Algorithm 3.8)reads through the *append_conList* and updates the global *ConnectionList*

Algorithm 3.7 Update Connection Information for *Migrated tasks*

```
procedure NewConnectionInfo(_map, new_ResourceList)
                        ▷ Obtain new Connection information for Migrated Resources
    _conList ← _map.pn.connectionList                              ▷ A local copy
    _mResList ← new_ResourceList
    Create append_conList
    i ← 1
    j ← 1
    repeat
        t_RName ← _mResList[i].Name
        repeat
            org ← _conList[j].org.Name
            trg ← _conList[j].trg.Name
            if (org ∋ t_RName) ∥ (trg ∋ t_RName) then
                l_con ← _conList[j]
                Update l_con
                if trg ≡ FIFO then
                    Update <Process> l_con.org
                    Update <Channel> l_con.trg
                else
                    Update <Channel> l_con.org
                    Update <Process> l_con.trg
                end if
                Append l_con to append_conList
            end if
            increment j
        until j ⩽ Con_N                              ▷ Con_N = _conList.length
        increment i
    until i ⩽ mRes_N                               ▷ mRes_N = _mResList.length
end procedure
```

Algorithm 3.8 Add New Connections to ConnectionList

```
procedure UpdateGlobalConnectionList(append_conList)
                                          ▷ Append Connections to Global List
    _conList ← _map.pn.connectionList                     ▷ Local copy for Update
    i ← 1
    repeat
        t_Con ← append_conList[i]
        if t_Con ∉ _conList then
            Append t_Con to _conList
        end if
        increment i
    until i ⩽ N_C                              ▷ N_C = append_conList.length
    _map.pn.connectionList ← _conList
                                           ▷ Global ConnectionList Updated
end procedure
```

3.6.7 Database Update for Remapped/Migrated Channels

This section provides the algorithmic approach for creating and updating the DAL/DOL database with additional *migrated channels*).

Addition of new *tasks/processes* require new *channel* connectivity either with the existing processes or with new processes. In the this implementation, the addition of new *channels* to the

Algorithm 3.9 Collect New Channel Elements (Table 3.4)

Require: $append_chList$
  procedure $NewChannelCreationFrom(_map, _channel, mode, type)$
                                                  ▷ Create a new Channel due to Migration Resources
    Create $new_channel :: new_channel \equiv _channel$
    $chName \leftarrow _channel.Name$
    if $mode \equiv$ **Migrate then**
      $new_channel.Name \leftarrow chName_Migrate$
      If required Update <Process> $new_channel.Origin$
      If required Update <Process> $new_channel.Target$
      Update $new_channel.portList$
    end if
    Append $new_channel$ **to** $append_chList$
  end procedure

  procedure $UpdateGlobalChannelList(append_chList)$
                                                  ▷ Append Channels to Global List
    $_chList \leftarrow _map.pn.channelList$ ▷ Local copy for Update
    $i \leftarrow 1$
    repeat
      $t_Ch \leftarrow append_chList[i]$
      if $t_Ch \notin _chList$ **then**
        Append t_Ch **to** $_chList$
      end if
      increment i
    until $i \leqslant N_{\mathcal{F}}$ ▷ $N_{\mathcal{F}} = append_chList.length$
    $_map.pn.channelList \leftarrow _chList$ ▷ Global *ChannelList* Updated
  end procedure

ChannelList is performed during new *connection* creation and updates.

The Algorithm 3.9 describes the creation of new *channel* based in *_map* information and existing *_channel* entity. The algorithm describing creation of new channels is governed by *mode* and *type*. This is done for optimization. The input quantity *mode* is used to distinguish the creation of *channel* for *task migration*. During *task migration* mode of channel creation, a copy of *_channel* is created called a *new_channel* with the channel name as *_channel.Name_Migrate*. If the destination process is migrated/remapped to some other processor/tile, the process information of the target process is extracted from the *ProcessList* and added to the *_channel.Target*. The same is valid for origin process as well. The input and output ports are updated with new *connection* information. Once all the updates are done in the *new_channel* element, it is added to the global *ChannelList* (procedure $UpdateGlobalChannelList()$).

Updates to either origin or target or both resources is performed based on the mode and type of channel creation. The entries in Table 3.4 is used to decide which resource/resources need to updated with new entries. The same principle is applied when port entries in the *new_channel.PortList* are updated. The type of channel created to support process remapping/migration is specified in Table 3.4. Either the source or destination or both can be migrated. The *Channel.portList* is accordingly updated with new connection information and process information. After all successful updates, the channel are added to the global *ChannelList*. In principle, this operation is done after every successful new channel creation.

Algorithm 3.10 Update Port Multiplexer Information

```
procedure UpdatePortMultiplexerData(_map)
                                        ▷ Update the Mux information for Process Ports
    _ChIter ← _map.pn.ChannelList
    _Ch ← _map.pn.ChannelList
    i ← 1
    j ← 1
    repeat
        tmpCh ← _ChIter[i]
        tmpChN ← tmpCh.Name
        tmpChBn ← tmpCh.Basename
        tmpOrgP ← _ChIter[i].Origin.Name
        tmpTrgP ← _ChIter[i].Target.Name
        repeat
            tCh ← _Ch[j]
            tChN ← tCh.Name
            tChBn ← tCh.Basename
            if tChN ≠ tmpChN && tChBn == tmpChBn then
                tOrgP ← _Ch[j].Origin.Name
                tTrgP ← _Ch[j].Target.Name
                if tTrgP.Name == tmpTrgP.Name then
                    k ← 1
                    tPort ← tTrgP.PortList
                    repeat
                        tPrtRes ← tPort[k].Resource
                        if tPort[k].isInPort && tPrtRes ≡ tTrgP then
                            Add* tmpOrgP to tPort[k].MuxProcess
                            Add* tmpCh.Name to tPort[k].MuxCh
                            Add* tmpOrgP(State) to tPort[k].MuxState
                            increment* tPortList[k].InMuxCnt
                            Add tOrgP to tPort[k].MuxProcess
                            Add tCh.Name to tPort[k].MuxCh
                            Add tOrgP(State) to tPort[k].MuxState
                            increment tPortList[k].InMuxCnt
                        end if
                        increment k
                    until k < PortList_N
                end if
            end if
            increment j
        until j < tCh_N
        increment i
    until i < tChIter_N
end procedure
```

3.6.8 Database Update for Process Port Mux/Demux blocks

The value of Mux/Demux count is used to determine if a multiplexer or a de-multiplexer block is required at the input or output port level. The default value of Mux/Demux count is zero, which signifies that neither a multiplexer nor a de-multiplexer is required. On the other hand, if more than one destination process is connected to given process. In addition to mux/demux count, the port also contains a list of process names, Active FSM states and channel names for assisting in code generation phase for different DAL visitors. In this work, the code generation phase is explained for the distributed SystemC visitor called *hdsd* (in Section 5).

The updated to port data structure (port structure in Figure 3.11, element in bold are the newly added element in DOL/DAL with respect to legacy DOL) with respect to multiplexer and de-multiplexer related information is explained in two phases. The first phase explains the multiplexer related data updates while the second phase deals with de-multiplexer related information. For simplicity the procedures related to multiplexer and de-multiplexer updates are separately[†] defined as *UpdatePortMultiplexerData()* and *UpdatePortDemultiplexerData()* respectively.

The Algorithm 3.10 is used to update the multiplexer related information of the input port entry in *PortList* element (Figure 3.11) of a process entity. The Algorithm with time complexity of $O(n^2)$ loops on the *processNetwork.ChannelList* and queries *mapping* table to obtain the information related to the given Port.Mux entities. In the algorithm, the two channels (*tmpCh* and *tCh*) are compared and a channel is selected such that the Channel.Basename for both the channels are same but the Name entry is different. Once the channel is selected ($_Ch[j]$), the origin and target resource name is extracted namely $tOrgP$ and $tTrgP$. The extracted target resource name is compared to channel target resource (from outer-loop namely, $_ChIter[i].Target.Name$) for equivalence. If processes name are identical, the PortList for the extracted target resource (process) is obtained and for every input port the element resource (*Port.Resource*) is extracted ($tPrtRes$). The target resource of the selected channel is again compared with the input port resource. If equivalent, the element related to multiplexer has to be updated with new values. As mentioned before, the initial value of the counter is zero and hence the first update is a special update (shown as **Add*** in the Algorithm) followed by normal updates (shown as **Add** in the Algorithm). In the special update (only called once), the data obtained from the outer loop is also added namely origin process (*tmpOrgP*), channel (*tmpCh*[*i*]) and Active state of origin process is added to the port database. The Active FSM states of a process is obtained by querying the *ComputationalBindingList* entity in *_map* data structure with process name. During normal updates, the data obtained from inner repeat loop is used for update. The entries added to the port sub-entities are origin process ($tOrgP$), channel (tCh) and Active FSM states, each corresponding to the information obtained the *ChannelList* ($_Ch$).

The Algorithm 3.11 is used to update the de-multiplexer information of the output port element (Figure 3.11) of a process entity. The basic software-flow of this procedure is same with small change in comparison value and update values. The entries in ChannelList namely *_ChIter* and *_Ch* are compared (same as in multiplexer Updates). Once the channel is selected ($_Ch[j]$), the origin resource name is extracted namely $tOrgP$. The extracted origin resource name is compared to channel origin resource (from outer-loop namely, *_ChIter[i].Origin.Name*) for equivalence. If processes name are same, the PortList for the extracted origin process resource is obtained and for every output port the sub-element resource (*Port.Resource*) is extracted ($tPrtRes$). The extracted origin resource of the channel ($_Ch[j]$) is again compared to the output port resource. If equivalent, the port element related to de-multiplexer is ready to be updated with new values. The representation of first update as a special update and normal updates follows the same principle (special update as **Add*** and normal update as **Add** in the Algorithm). In the special update (only called once), the data obtained from the outer loop is also added namely target process (*tmpTrgP*), channel (*tmpCh*[*i*]) and Active state of target process is added to respective sub-entities of the port database. In normal updates, the entries like target process ($tTrgP$), channel (tCh) and Active FSM states correspond to the information obtained from the *ChannelList* ($_Ch[j]$) and are added to the respective sub-entities.

[†]Due to Algorithm representation and comprehensibility in report, the actual implementation code was broken down into two distinct algorithms.

Algorithm 3.11 Update Port De-multiplexer Information

```
procedure UpdatePortDemultiplexerData(_map)
                                    ▷ Update the Demux information for Process Ports
    _ChIter ← _map.pn.ChannelList
    _Ch ← _map.pn.ChannelList
    i ← 1
    j ← 1
    repeat
        tmpCh ← _ChIter[i]
        tmpChN ← tmpCh.Name
        tmpChBn ← tmpCh.Basename
        tmpOrgP ← _ChIter[i].Origin.Name
        tmpTrgP ← _ChIter[i].Target.Name
        repeat
            tCh ← _Ch[j]
            tChN ← tCh.Name
            tChBn ← tCh.Basename
            if tChN ≠ tmpChN && tChBn == tmpChBn then
                tOrgP ← _Ch[j].Origin.Name
                tTrgP ← _Ch[j].Target.Name
                if tOrgP.Name == tmpOrgP.Name then
                    k ← 1
                    tPort ← tOrgP.PortList
                    repeat
                        tPrtRes ← tPort[k].Resource
                        if tPort[k].isOutPort && tPrtRes ≡ tTrgP then
                            Add* tmpTrgP to tPort[k].DemuxProcess
                            Add* tmpCh.Name to tPort[k].DemuxCh
                            Add* tmpTrgP(State) to tPort[k].DemuxState
                            increment* tPortList[k].OutDemuxCnt
                            Add tTrgP to tPort[k].DemuxProcess
                            Add tCh.Name to tPort[k].DemuxCh
                            Add tTrgP(State) to tPort[k].DemuxState
                            increment tPortList[k].OutDemuxCnt
                        end if
                        increment k
                    until k < PortList_N
                end if
            end if
            increment j
        until j < tCh_N
        increment i
    until i < tChIter_N
end procedure
```

4

Fault Tolerance

4.1 Overview

This chapter contains the key concept involved in the creation and designing a fault tolerant system using the DAL middle-ware for EURETILE architecture and for generic applications executing on it. The section in this chapter describes the inclusion of fault tolerance concept on to *dynamic application* behavior.

4.2 Problem Definition

The problem definition is listed below:

- Fault tolerance at application level to prevent system from stalling in case of errors and faults.

4.3 Fault Definition and Recovery

The initial work in developing DAL framework is focused towards detecting faults like non-responsive processor core. This can happen at run-time due to transistor level faults like soft-errors. The scope of this work is limited to recover the system from a non-responsive processor. The work does not include the implementation and use of detect faults mechanism for MPSoC.

The fault recovery can be achieved using either of the two fault tolerance mechanisms:

- Static Fault Tolerance: This is a proactive mechanism. The application processes, channels and connections are replicated based on some predefined fault tolerance scenarios (Figure 4.1). New connections and channels are created to connect the original processes and duplicated processes (referred to as clones) and vice versa. In the event of fault, the master controller activates the clone processes by redirecting the control message on to the appropriate controller slave (Figure 4.2). This mechanism provides faster recovery from fault. The overall mean-time to recover is minimal. This mechanism can be adopted for

system critical applications. The Figure 4.1 and Figure 4.2 depict the the static fault tolerance setup and recovery mechanism in case of faulty processor. More details about static fault tolerance is explained later in the chapter.

- Dynamic Fault Tolerance: This is a reactive mechanism. The remapping of processes, channels and connections are initiated as a recovery mechanism after the fault has been identified and isolated. The Figure 4.3 shows a general method adopted for supporting dynamic fault tolerance. The fault recovery in case of dynamic fault tolerance is shown in Figure 4.4.

Other semi-dynamic mechanism can be derived using the above basic fault tolerance mechanism.

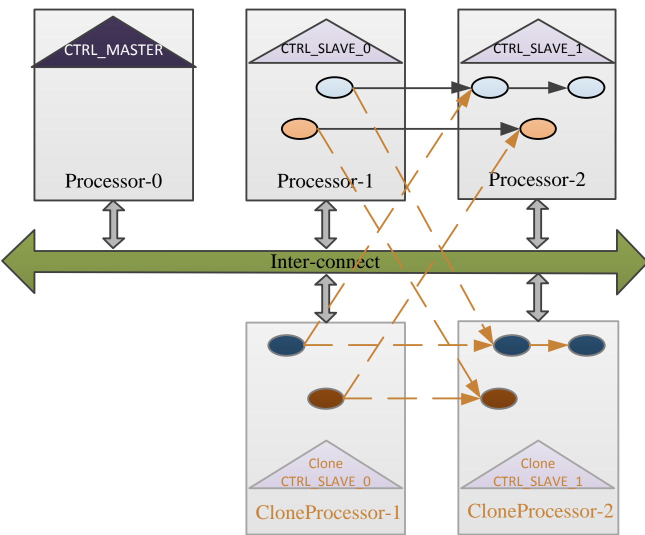


Figure 4.1: Static fault tolerance setup during code generation and initialization.

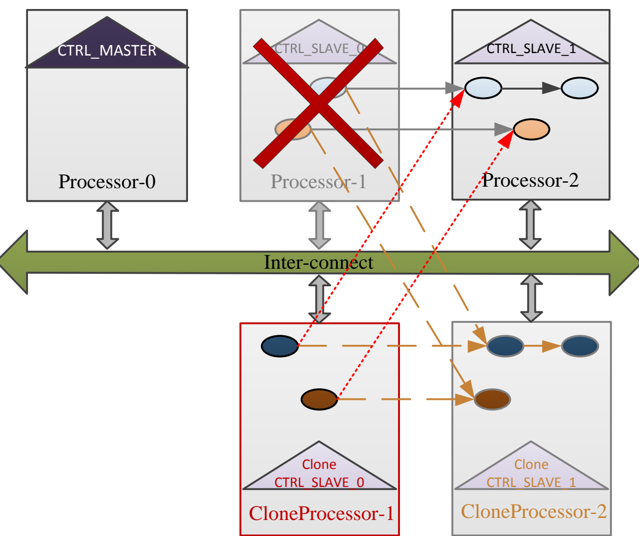


Figure 4.2: Recovery in static fault tolerance setup on the event of faulty processor.

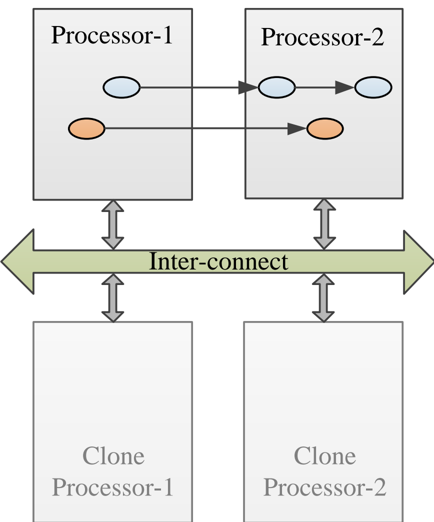


Figure 4.3: Dynamic fault tolerance initial setup.

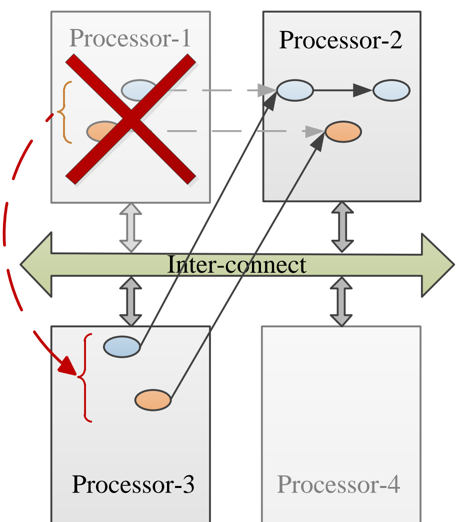


Figure 4.4: Recovery in system with dynamic fault tolerance support.

4.4 Fault Tolerance in DAL

Designing highly reliable and fault tolerant system would required processor and task redundancies such that mean time to recovery (MTTR) from a failure is kept as small as possible. One of the additional design requirement of the DAL middle-ware is to adapt the system to faults and remap the application behavior. This is termed as *remapping strategy* in context to DAL Middleware. Statically, it would be impossible and impractical to explore all possibilities related to fault and derive an optimal mapping solution to recover the system from the fault. A low overhead, fast on-line fault recovery strategy would be an optimal solution to this. The *remapping strategies* can be broadly classified into three categories namely Figure 4.5:

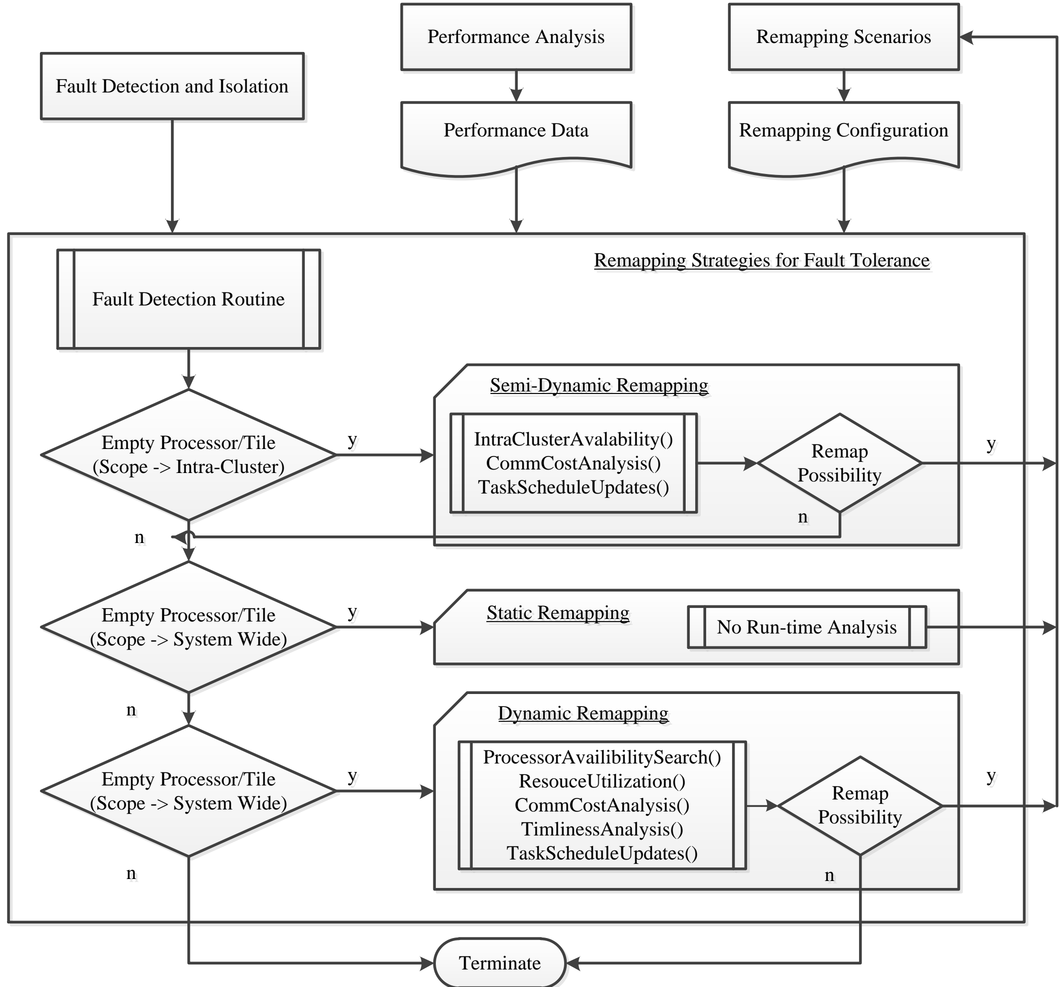


Figure 4.5: Graphical representation of Fault Tolerance Strategies using Flow Charts.

- Static Remapping Strategy: The policy can be adopted for system that execute highly critical applications and it is required to keep the value of MTTR as low as possible. There system should support seamless control from the fault processor to non-active processor. This policy can be adopted on system that have *redundant processors* for highly critical system applications (preferably for all the *slave processors*). The hardware architecture design and task mapping should be devised such that the communication cost in term of *packet trip time* should be approximately identical to the non-faulty case; for instance adopting identical and symmetrical design for clusters. The basic idea is to create *cloned*

application tasks on *clone processor*, similar to *original application tasks* on the *original processors*. Initially, the original application tasks mapped to original processor are activated, whereas the clone tasks on the clone processor are rendered inactive. The *master controller* maintains a table, *Active_Processor*, of active processors along with its corresponding clone processor information. The *master controller* also maintains a real-time database/table, *Channel_Process_Port*, of information pertaining to channels connected to origin process/task to target process/task. At run-time, the *master controller* queries the *Active_Processor Table* and *Channel_Process_Port Table* and decides the content of the *control message* and *active destination slave processor*. This solution provides a responsive low overhead fault-tolerant model of the system at the cost of higher architectural cost, cloned processor, and under-utilized system.

- Semi-Dynamic Remapping Strategy: Statically remapping entire processor/cluster would considerably increase the architecture cost of an under-utilized system. Instead of mapping all the application tasks, only the tasks pertaining to the faulty processor/tile (depending upon the fault type) can be mapped on to the under-utilized processor/tile or a idle processor/tile (preferably within same clusters). If the system is grouped into clusters, the *Cluster Level controller* queries the intra-cluster information and computes the remapping options of the *tasks* stalled due to faulty processor/tile. If the *Local Cluster controller* finds an option it remaps the *task* $\widehat{T_l}$ to the available *processors/tiles*. On the contrary, if the *Local Cluster controller* fails to find any valid remapping option, it sends control message to the *master controller* with the list of stalled tasks to initiate inter-cluster remapping processing. The inter-cluster remapping can possibly lead to increase in the *packet trip time*. Hence this strategy provides flexibility at the cost of decrease in system efficiency.

- Dynamic Remapping Strategy: In a Dynamic remapping strategy, the *task* $\widehat{T_l}$ computational remapping and the *channel* $\widehat{Ch_l}$ remapping related to communication is decided at global level based on lasted processor/tile performance and resource utilization. As the decision is made at the global level, the time required for run-time decision is considerably higher. This would also result in increase of the *communication* step-up time and data communication time (due to geometric positioning of the *application tasks*). The DAL applications are stopped and restarted during the failover-takeover phase. This model of remapping strategy offers higher flexibility at the cost of much higher run-time decision computation time, communication set-up time and communication time.

$$\widehat{PN}_i = [\widehat{T}_i, \widehat{Con_i}, \widehat{Ch_i}], \forall i \in \mathcal{N} \tag{4.1}$$

$$PN = \{PN_i \cup \widehat{PN}_i : \forall i \in \mathcal{N}\} \tag{4.2}$$

$$T = \{T_i \cup \widehat{T}_i \cup \overline{T}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\alpha}_{\mathcal{K}}\} \tag{4.3}$$

$$Con = \{Con_i \cup \widehat{Con_i} \cup \overline{Con}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\beta}_{\mathcal{K}}\} \tag{4.4}$$

$$Ch = \{Ch_i \cup \widehat{Ch_i} \cup \overline{Ch}_k : \forall i \in \mathcal{N}, \forall k \in \overline{\gamma}_{\mathcal{K}}\} \tag{4.5}$$

In case of first model *Statical Remapping Strategy*, the entire application task instance is replicated (clones) on to its corresponding clone slave processors. The Equation 4.1 describes the *cloned application process network*, $\widehat{PN}_i, \forall i \in \mathcal{N}$, information. The elements $\widehat{T}_i$, $\widehat{Con_i}$ and $\widehat{Ch_i}$ describe the *cloned application tasks*, *cloned connections* and *cloned channels* respectively. The aggregate collection of *application tasks*, T, can be described as a super set of *original tasks* T_i, *migrated tasks* $\overline{T}_k$ and *cloned tasks* $\widehat{T}_i$; using Equation 4.3. In a similar manner, the entire collection of

application connections, Con, is a super set of *original connections* Con_i, *migrated connection* $\overline{Con}_k$ and *cloned connections* $\widehat{Con}_i$; using Equation 4.4 and the *Software channels*, Ch is a super set of *original channels* Ch_i, *migrated channels* $\overline{Ch}_k$ and *cloned channels* $\widehat{Ch}_i$; using Equation 4.5. At *process network* level, the resultant PN is union of *original process network* PN_i and *cloned process network* $\widehat{PN}_i$ (Equation 4.2).

$$T = \{T_i \cup \widehat{T}_l \cup \overline{T}_k : \forall i \in \mathcal{N}, \forall l \in \widehat{\alpha}_{fail}, \forall k \in \overline{\alpha}_{\mathcal{K}}\} \tag{4.6}$$

$$Con = \{Con_i \cup \widehat{Con}_l \cup \overline{Con}_k : \forall i \in \mathcal{N}, \forall l \in \widehat{\beta}_{fail}, \forall k \in \overline{\beta}_{\mathcal{K}}\} \tag{4.7}$$

$$Ch = \{Ch_i \cup \widehat{Ch}_l \cup \overline{Ch}_k : \forall i \in \mathcal{N}, \forall l \in \widehat{\gamma}_{fail}, \forall k \in \overline{\gamma}_{\mathcal{K}}\} \tag{4.8}$$

In case of *Semi-dynamic and Dynamic Remapping Strategy*, a small subset of the entire application tasks instance is replicated on to its corresponding slave processors decided by the on-line system wide information and scheduling analysis. The elements $\widehat{T}_l$, $\widehat{Con}_l$ and $\widehat{Ch}_l$ describe the *cloned application tasks*, *cloned connections* and *cloned channels* respectively ($\forall l \in \widehat{\alpha}_{fail}$ and $\widehat{\alpha}_{final} \ll \mathcal{N}$, i.e. the number of faulty tasks). The aggregate collection of *application tasks* for *Semi-dynamic and Dynamic Remapping Strategies*, T, can be described as a super set of *original tasks* T_i, *migrated tasks* $\overline{T}_k$ and *Cloned tasks* $\widehat{T}_l$; using Equation 4.6. In a similar manner, the entire collection of *application connections*, Con, is a super set of *original connections* Con_i, *migrated connection* $\overline{Con}_k$ and *Cloned connections* $\widehat{Con}_l$; using Equation 4.7 and the *Software channels*, Ch is a super set of *original channels* Ch_i, *migrated channels* $\overline{Ch}_k$ and *cloned channels* $\widehat{Ch}_l$; using Equation 4.8.

4.5 Controller Clones

The concept of fault tolerance can be extended to slave controller tasks residing on processors designated as clone processors. The master controller decides to send control packets of appropriate slave based on some static control information. In addition to increase in run-time complexity, the system has a single point of failure; the entire system stalls if the *master controller* fails (without fault tolerance at *master controller* level). Adding a fault tolerant *master controller* would incorporate system-level database synchronization between *master controller* and $\widehat{Master}$ *controller* (clone); even higher algorithmic complexity.

The Equation 3.11 and Equation 3.12 describes in details the elements $Ctrl_PN^{Master}$ and $Ctrl_PN_j^{Slave}$ in term of its respective *controller tasks* ($Ctrl_T^{Master}$ / $Ctrl_T_j^{Slave}$), the *connections* defined between the *controller task* ($Ctrl_Con^{Master}$ / $Ctrl_Con_j^{Slave}$) and the *software channels* connecting the *ports* of the *controller tasks* ($Ctrl_Ch^{Master}$ / $Ctrl_Ch_j^{Slave}$).

$$\begin{aligned} \widehat{Ctrl_PN}_j^{Slave} =& [\widehat{Ctrl_T}_j^{Slave}, \widehat{Ctrl_Con}_j^{Slave}, \widehat{Ctrl_Ch}_j^{Slave}] \\ & \forall j \in \mathcal{J}_{clone}, \mathcal{J}_{clone} = \mathcal{J} \end{aligned} \tag{4.9}$$

One of the aspect of Fault Tolerant DAL middle-ware is that it requires the cloning of *application task* on its respective cloned *processors*. The *slave process network controller* residing on the *cloned processor* is termed as $\widehat{Ctrl_PN}_j^{Slave}$ and the *cloned slave controller task* is pointed by $widehatCtrl_T_j^{Slave}$. The *cloned slave process network controller*, Equation 4.9, is defined in the similar fashion as the *slave process network controllers* (Equation 3.12).

Algorithm 4.1 Updating Computational Binding Information for *cloned task/process*

```
procedure ObtainNewTaskInfo(_map)
                                    ▷ Obtain Task from the Computational Binding List
    local_pList ← _map.pn.processList                                 ▷ A local copy
    local_cBindList ← _map.compBindList
    Create append_pList
    i ← 1
    j ← 1
    repeat
        t_bPName ← local_cBindList[i].Name
        if (t_bPName ∋ Clone) then
            repeat
                _PName ← local_pList[j].Name
                if _PName ∋ t_bPName then
                    if _PName ≠ t_bPName then
                        Create new_p = local_pList[j]
                        new_p.Name = t_bPName
                        Append new_p to appened_pList
                    end if
                end if
                increment j
            until j ⩽ T_N                               ▷ T_N = local_pList.length
        end if
        increment i
    until i ⩽ B_N                                   ▷ B_N = local_cBindList.length
end procedure

procedure UpdateGlobalProcessList(append_pList)
                                               ▷ Append Processes to Global List
    _procList ← _map.pn.processList                        ▷ Local copy for Update
    i ← 1
    repeat
        t_Proc ← append_pList[i]
        if t_Proc ∉ _procList then
            Append t_Proc to _procList
        end if
        increment i
    until i ⩽ N_L                                 ▷ N_L = append_procList.length
    _map.pn.processList ← _procList                                    ▷ Updated
end procedure
```

4.6 Implementation

This section deals with different aspects of DAL database update of visitor based code generation and execution.

4.6.1 Database Update for Cloned Processes

This section provides the details in terms of algorithmic approach adopted for updating the DAL/DOL database for *code generation* process of a given *visitor*. The method of database update is governed by the addition of *cloned tasks/processes* to the process network database. The information about *cloned task/process* is added to the DAL/DOL database in two stages.

Algorithm 4.2 Updating the elements in new *Process* instance

Require: A $_processIns$ instance to be updated
  procedure $UpdateProcessContents(_processor, _portList)$
  ▷ Update Process sub-entities
    $_processIns.Processor \leftarrow _processor$
    $_processIns.portList \leftarrow _portList$
  end procedure

- In the first phase, the binding information is obtained from the *_map* data structure. The *Process.Name* information in the bindingList is searched for sub-string like **Clone**. In Algorithm 4.1, the first procedure ($ObtainNewTaskInfo()$) is used for obtaining a list of *ProcessName* from the *ComputationalBindingList*. Whenever a *ProcessName* satisfies the previous condition (with keyword **Clone**) and the *ProcessName* matches closest to any *Process.name* in the *ProcessNetwork* (original) (Figure 3.11), a new local copy (non-linked) of the *Process* instance (e.g. *tmp_Process*) is created and the *name* entity is modified with *ProcessName*.

  $$tmp_Process.name \leftarrow ProcessName$$

  The process instance *tmp_Process* is added to *processNetwork.ProcessList* if the entity is not present in the list (using the $UpdateGlobalProcessList()$ in Algorithm 4.1). For efficiency, the addition of new elements to the *ProcessList* is done just once after all the *tmp_Process* have been added to a Vector of processes.

- In the second phase of *process* updates, the structural elements like the *port* information is updated. This update is done when the new *connection* and *channel* information are created and added to the *ConnectionList* and *ChannelList* respectively. In the following Sections, the procedural call to $UpdateProcessContents()$ [Algorithm 4.2] is made whenever a new *connection* is added to *ConnectionList* and *ChannelList*.

4.6.2 Database Update for Clone Connections

The binding information of the *process port* to a *channel port* is stored in data structure called *connection*. The instances of connection is broadly classified to

- Origin Connection: The *origin* resource is a *process* and *target* resource is a *channel*. Hence at database level, the following information is stored:
  - $connection_ins.origin_resource.type \leftarrow NULL$
  - $connection_ins.target_resource.type \leftarrow FIFO$
- Target Connection: The *origin* resource is a *channel* and *target* resource is a *process*. The following information stored is at the connection instance level in the database.
  - $connection_ins.origin_resource.type \leftarrow FIFO$
  - $connection_ins.target_resource.type \leftarrow NULL$

Unlike the connection types in task remapping/migration, the cloning specification supports six types of *connections*. An example of all possible connections related to cloning is shown in Table 4.1. This information is the basis for addition of new connection entries to the *ConnectionList*.

For simplicity, the new connection entry creation is bifurcated into two different procedures namely $CloneOrgConnectionInfo()$ (Algorithm 4.4) and $CloneTrgConnectionInfo()$ (Algorithm 4.5). The former is used for generating new origin connections and latter is used for

Algorithm 4.3 Add New Connections to ConnectionList

procedure $UpdateGlobalConnectionList(append_conList)$
▷ Append Connections to Global List
  $_conList \leftarrow _map.pn.connectionList$ ▷ Local copy for Update
  $i \leftarrow 1$
  repeat
    $t_Con \leftarrow append_conList[i]$
    if $t_Con \notin _conList$ **then**
      Append t_Con **to** $_conList$
    end if
    increment i
  until $i \leqslant N_{\mathcal{C}}$ ▷ $N_{\mathcal{C}} = append_conList.length$
  $_map.pn.connectionList \leftarrow _conList$
▷ Global *ConnectionList* Updated
end procedure

Table 4.1: Clone Connection and Channel Types

Type	Origin Process (Origin Connection)	Channel	Target Process (Target Connection)
1	A (tmp_AC1)	tmp_C1	$Clone_B$ (tmp_C1B)
2	$Clone_A$ (tmp_Clone_AC1)	tmp_Clone_C1	B (tmp_Clone_C1B)
3.	$Clone_A$ ($Clone_AC1$)	$Clone_C1$	$Clone_B$ ($Clone_C1B$)

generating new target connections for supporting static fault tolerance mechanism in DAL Framework.

As mentioned, the Procedure $CloneOrgConnectionInfo()$ handles the new connection creation for *clone processes* and *clone channels*. Like the Procedure $NewConnectionInfo()$ (Algorithm 3.7), the Algorithm 4.4 take $_map$ and $new_ResourceList$ as the input data to generate new connections. Unlike the Algorithm 3.7, the Algorithm 4.4 considers only the origin resource Name for finding the occurrence of sub-string of the *org* (entity:*Connection.Origin.Name*) in *t_cName* (entity: *new_ResourceList[i].Name*; $\forall i \in new_ResourceList.length()$). On successful match of sub-string, a new local copy of *new_con*,(non-linked) copy of connection instance (*_conList[j]*; $\forall j \in _map.pn.ConnectionList.length()$) is created. As the connection type for *cloned process* has three categories, Table 4.1, three new connections will be added to the *ConnectionList*. A brief description of New connections is mentioned below:

- Origin Connection Type-1: A new connection is created namely, with a Connection.Name as *tmp_<OriginConnection.Name>*. The entity *BaseName* would remain the same*. As the origin resource in connection entity is of type process, the contents of origin entity is retained; only the output port information containing the PeerResource (i.e. *_conList[j].portList.port[l].PeerResource* ≡ *_conList[j]*) is updated to *new_con*. The target resource, of type channel, is newly created, namely *tmp_<ChannelName>* (Algorithm 4.6). The target process entities in channel have to be updated to process with Name *Clone_<Process.Name>*. Additionally, the PeerResource entity for the output port should be modified to *new_con*. After all these updates, the *new_con* is added to the *append_conList*.

*As seen in the Figure 2.9, an application process (C code with process-specific routines like initialize(), fire(), etc) is wrapped inside a process wrapper in SystemC Distributed application code. The process wrapper is built over the BaseName to avoid memory overhead for instances of process with different *ProcessName* but with same functionalities.

Algorithm 4.4 Update Origin Connections for *Clone tasks*(Table 4.1)

```
procedure CloneOrgConnectionInfo(_map, new_ResourceList)
                                        ▷ Obtain new Connection information for Clone Resources
    _conList ← _map.pn.connectionList
    _cResList ← new_ResourceList
    Create append_conList
    i ← 1
    j ← 1
    repeat
        t_cName ← _cResList[i].Name
        repeat
            org ← _conList[j].org.Name
            if org ∋ t_cName then
                Create new_con ← _conList[j]
                                        ▷ Type 1: Connection ≡ tmp_ < con_name
                                        ▷ Type 2: Connection ≡ tmp_Clone_ < con_name
                                        ▷ Type 3: Connection ≡ Clone_ < con_name
                Create <Process> ≡ new_con.org_p
                                        ▷ Type 1: New Process ≡ < proc_name >
                                        ▷ Type 2: New Process ≡ Clone_ < proc_name >
                                        ▷ Type 3: New Process ≡ Clone_ < proc_name >
                Create <Channel> ≡ new_con.trg_ch
                                        ▷ Type 1: New Channel ≡ tmp_ < ch_name >
                                        ▷ Keep OriginProc; Update TargetProc
                                        ▷ Type 2: New Channel ≡ tmp_Clone_ < ch_name >
                                        ▷ Update OriginProc; Keep TargetProc
                                        ▷ Type 3: New Channel ≡ Clone_ < ch_name >
                                        ▷ Update OriginProc; Update TargetProc
                new_con.org ← new_con_org_p
                new_con.trg ← new_con_trg_ch
                Append new_con to append_conList
            end if
            increment j
        until j ⩽ Con_N                         ▷ Con_N = _conList.length
        increment i
    until i ⩽ cRes_N                            ▷ cRes_N = _cResList.length
end procedure
```

- Origin Connection Type-2: Herein for clone processes, a new local instance of connection is created with a Connection.Name as *tmp_ Clone_ <OriginConnection.Name>*. As in Type 1, the entity *BaseName* is unaltered. Unlike the Type-1, the origin resource in connection entity (of type process) is updated to process with Process.Name as *Clone_ <OriginProcess.Name>*. The resource in output port entity is updated to new process (with Process.Name ≡ *Clone_ <OriginProcess.Name>*). The PeerResource in the output port entity (i.e. *_ conList[j].portList.port[l].PeerResource* ≡ *_ conList[j]*) is also updated to *new_ con*. A new instance of target resource of connection *new_ con* (of type channel) is created with Channel.Name as *tmp_ Clone_ <ChannelName>* (Algorithm 4.6). The origin process entities in channel has to be updated to process with name *Clone_ <OriginProcess.Name>*. Additionally, the PeerResource entity for the input port is modified to *new_ con*. Post the resource entry updates, the *new_ con* is added to the *append_ conList*.

- Origin Connection Type-3: As in other types of source connections, in case of clones a new local connection instance with a Connection.Name as *Clone_ <OriginConnection.Name>* is created. The *BaseName* entity is kept unaltered for optimization. The origin re-

Algorithm 4.5 Update Target Connections for *Clone Tasks*(Table 4.1)

```
procedure CloneTrgConnectionInfo(_map, new_ResourceList)
                              ▷ Obtain new Connection information for Clone Resources
    _conList ← _map.pn.connectionList
    _cResList ← new_ResourceList
    Create append_conList
    i ← 1
    j ← 1
    repeat
        t_cName ← _cResList[i].Name
        repeat
            trg ← _conList[j].trg.Name
            if trg ∋ t_cName then
                Create new_con ← _conList[j]
                                        ▷ Type 1: Connection ≡ tmp_ < con_name
                                        ▷ Type 2: Connection ≡ tmp_Clone_ < con_name
                                        ▷ Type 3: Connection ≡ Clone_ < con_name
                Create <Channel> ≡ new_con.org_ch
                                        ▷ Type 1: New Channel ≡ tmp_ < ch_name >
                                        ▷ Keep OriginProc; Update TargetProc
                                        ▷ Type 2: New Channel ≡ tmp_Clone_ < ch_name >
                                        ▷ Update OriginProc; Keep TargetProc
                                        ▷ Type 3: New Channel ≡ Clone_ < ch_name >
                                        ▷ Update OriginProc; Update TargetProc
                Create <Process> ≡ new_con.trg_p
                                        ▷ Type 1: New Process ≡ < proc_name >
                                        ▷ Type 2: New Process ≡ Clone_ < proc_name >
                                        ▷ Type 3: New Process ≡ Clone_ < proc_name >
                new_con.org ← new_con_org_ch
                new_con.trg ← new_con_trg_p
                Append new_con to append_conList
            end if
            increment j
        until j ⩽ Con_N                          ▷ Con_N = _conList.length
        increment i
    until i ⩽ cRes_N                             ▷ cRes_N = _cResList.length
end procedure
```

source in connection entity (of type process) is updated to process with Process.Name as *Clone_ <OriginProcess.Name>*. The resource entity in output port entity is updated to process with Process.Name ≡ *Clone_ <OriginProcess.Name>*. In addition to resource entry update, the PeerResource in the output port entity(i.e. *_conList[j].portList.port[l].PeerResource* ≡ *_conList[j]*) is also updated to *new_con*. A new instance of target resource of connection *new_con* (of type channel) is created with Channel.Name as *Clone_ <ChannelName>* (Algorithm 4.6). The origin and target process entities in channel is updated to processes with Name *Clone_ <OriginProcess.Name>* and *Clone_ <TargetProcess.Name>* respectively. Additionally, the PeerResource entity for the input port and output port is modified to *new_con*. After all these updates, the connection instance, *new_con* is added to the *append_conList*.

After all the source connection updates and addition of new connection entires to the *append_ConList*, the routine (Algorithm 4.3) is called for appending the entries to global *ConnectionList*.

The Procedure *CloneTrgConnectionInfo*() handles the new connection creation for clone channels and clone process. This Algorithm 4.5 requires details of *_map* and *new_ResourceList* for generating new connection instances. As this Algorithm is specifically created for target connections, only the element with the target resource name is used for searching the connection instances such that the *trg* (entity:*Connection.Origin.Name*) is contained in *t_cName* (*new_ResourceList[i].Name*; $\forall i \in$ *new_ResourceList.length*()) (refer to Algorithm 4.5). On successful sub-string match, like source connection, a new local *new_con* (non-linked) copy of connection instance (*_conList[j]*; $\forall j \in$ *_map.pn.ConnectionList.length()*) is created. As indicated in Table 4.1, three new connections will be added to the *ConnectionList*. The brief description of new connections is mentioned below:

- Target Connection Type-1: A new connection with a Connection.Name as *tmp_<TargetConnection.Name>* is created. The *BaseName* entity information remains unmodified. The origin resource in connection entity is of type channel. If the channel with channel.Name as *tmp_<ChannelName>* does not exist in the *ChannelList*, a local copy of the channel entity is created. The contents of origin process is retained whereas the target process is updated to process with process name as *Clone_<TargetProcess.Name>*. The port information is accordingly updated. In the target connection type, the target resource is a process. A target resource is updated to process with process name as *Clone_<TargetProcess.Name>*. Like source connection updates, the connection instance, *new_con* is added to the *append_conList*.
- Target Connection Type-2: For the target connection type-2, a new local instance of connection is created with a Connection.Name as *tmp_Clone_<TargetConnection.Name>*. The origin resource in connection entity (of type channel) is updated to channel, *tmp_Clone_<ChannelName>*. If this channel is already created, the content of origin resource is replaced with the channel, else a new channel is created. The origin and target resource is updated along with port informations. The target resource of the connection *new_con* is left unaltered with an exception. The PeerResource needs to be updated with *new_con* information for connection input port. Since the *new_con* is to be appended to *ConnectionList*, it is added to a list of new connections, the *append_conList*.
- Target Connection Type-3: As in other types of target connections, for the type-3 connection for scenario with clones, a new local connection instance with a Connection.Name as *Clone_<TargetConnection.Name>* is created. The *BaseName* entity is kept unaltered for optimization. The target resource in connection entity (of type process) is updated with process information containing Process.Name as *Clone_<TagetProcess.Name>*. The resource entity in input port entity is updated to process with Process.Name $\equiv$ *Clone_<TargetProcess.Name>*. In addition to resource entry update, the PeerResource in the input port entity(i.e. *_conList[j].portList.port[l].PeerResource* $\equiv$ *_conList[j]*) is also updated to *new_con*. A new instance of origin resource of connection *new_con* (of type channel) is created with channel Name as *Clone_<ChannelName>* (Algorithm 4.6). The origin and target process entities in channel is updated to processes with Name *Clone_<OriginProcess.Name>* and *Clone_<TargetProcess.Name>* respectively. Additionally, the PeerResource entity for the input port and output port is modified to *new_con*. The connection instance, *new_con* is then added to the *append_conList*.

After all the target connection updates and addition of new connection entires to the *append_ConList*, the routine (Algorithm 4.3) is called for appending the entries to global *ConnectionList*.

4.6.3 Database Update for Clone Channels

Addition of new *tasks/processes* require new *channel* connectivity either with the existing processes or with new processes. In the this implementation, the addition of new *channels* to the *ChannelList* is performed during new *connection* creation and updates.

Algorithm 4.6 Collect New Channel Elements (Table 4.1)

```
Require: append_chList
  procedure NewChannelCreationFrom(_map, _channel, mode, type)
                              ▷ Create a new Channel due to Migration or Clone Resources
     Create new_channel :: new_channel ≡ _channel
     chName ← _channel.Name
     if mode ≡ Clone then                      ▷ chName may also contain Migrate
        if type ≡ Type 1 then
           new_channel.Name ← tmp_chName
           OR
           new_channel.Name ← tmp_chName_Migrate
           Update <Process> new_channel.Target
        else if type ≡ Type 2 then
           new_channel.Name ← tmp_Clone_chName
           OR
           new_channel.Name ← tmp_Clone_chName_Migrate
           Update <Process> new_channel.Origin
        else                                                   ▷ type ≡ Type 3
           new_channel.Name ← Clone_chName
           OR
           new_channel.Name ← Clone_chName_Migrate
           Update <Process> new_channel.Origin
           Update <Process> new_channel.Target
        end if
        Update new_channel.portList
     end if
     Append new_channel to append_chList
  end procedure

  procedure UpdateGlobalChannelList(append_chList)
                                              ▷ Append Channels to Global List
     _chList ← _map.pn.channelList                        ▷ Local copy for Update
     i ← 1
     repeat
        t_Ch ← append_chList[i]
        if t_Ch ∉ _chList then
           Append t_Ch to _chList
        end if
        increment i
     until i ⩽ N_F                              ▷ N_F = append_chList.length
     _map.pn.channelList ← _chList                ▷ Global ChannelList Updated
  end procedure
```

The Algorithm 4.6 describes the creation of new *channel* based in *_map* information and the *_channel* using *mode* and *type* (for optimization). The input quantity *mode* is used to distinguish between the creation of *channel* for *task migration* and *task cloning*. The element *type* is used to differentiate the three types of channels required for *task cloning*. During *task migration* mode of channel creation, the a copy of *_channel* is created called a *new_channel* with the channel name as *_channel.Name_Migrated*. If the destination process is migrated (Section 3.5) to some other processor/tile, the process information of the target process is extracted from the *ProcessList* and added to the *_channel.Target*. The same is valid for origin process as well. The input and output ports are updated with new *connection* information. Once all the updates are done in the *new_channel* element, it is added to the global *ChannelList* (procedure *UpdateGlobalChannelList*()).

The creation of new channel for *task cloning* also follows a similar procedure with few added steps (like Algorithm 3.9). Updates to either origin or target or both resources is performed based on the mode and type of channel creation. The entries in Table 4.1 is used to decide as to which resource/resources need to updated with new entries. The same principle is applied when port entries in the *new_channel.PortList* are updated. If the channel to be created is of type 1, the channel is named as *tmp_ <ChannelName>* and the target resource is modified. For type 2 channel creation, the *new_channel* is named as *tmp_Clone_ <ChannelName>* and the origin resource is modified. In case of type 3, both the origin and target are *clone process* and hence the corresponding entries in the *new_channel* is updated. The channel is named as *Clone_ <ChannelName>*. The Channel.portList is accordingly for all three types updated with new connection information and process information. After all successful updates, the channel are added to the global ChannelList in process network data structure. In principle, this operation is done after every successful new channel creation. The benefit of this method is the uniform creation algorithm for either *clone migrated channels* or only *clone channels*. On the other hand, the recurrent updates to *ChannelList* with new channel information is run-time overhead during database creation.

4.6.4 Database Update for Process Port Mux/Demux blocks

The value of Mux/Demux count is used to determine if a multiplexer or a de-multiplexer is required at the input or output port level. The default value of Mux/Demux Count is kept as zero as mentioned in section 3.6.8. The updates to the port based Mux/Demux blocks namely, the count value, process names, channel names and FSM state names can be achieved using the Algorithm 3.10. For more details on the algorithmic description, refer to Section 3.6.8.

5
Simulator

This chapter uses the database created and updated using algorithms mentioned in previous Chapters 3,4 to create a simulation environment. This chapter describes in detail the implementation aspects of the concepts pertaining to DAL requirements for a given visitor, Figure 3.4. This chapter describes the usage of updated the DAL/DOL database with *migrated* and *cloned* information to implement the *automated code generation* mechanism for a *visitor*[*]. The scope of *automated code generation* for *visitor* in this work is limited to *hdsd* - *(hardware dependent software distributed)*. The *hdsd* is built using SystemC Distributed library, Section 2.3. For easy of comprehension refer to the data entity relationship model of DAL/DOL middle-ware, Figure 3.11.

5.1 DAL Visitor Overview

The DAL/DOL middle-ware is used to generate a range of output executables on a variety of platforms like MPARM [33], Cell Broadband Engine [4], etc. The software work-flow for DAL is shown in Figure 3.4. The Figure 5.1 describes the relational work-flow for the *hdsd* visitor.

The visitor for distributed SystemC simulation i.e. *hdsd*, is governed by the information present in mapping *_map*, process network specification *_pn*, architectural specification *_arch* and FSM-related database *_fsm*. The steps involved in code generation is as follows:

- The *hdsd* directory creation as per User-defined input.
- For compilation process, a Makefile is generated.
- The creation of architectural specific SystemC application.
- Process wrapper creation (*header* and *cpp* file).
- Creation of linux script file for distribution of SystemC executable.
- Creation of controller wrapper code for all SystemC Distributed executables.

The subsection mentioned below contains the major changes made to the SystemC application code (specific to DAL middle-ware).

[*]In this chapter the term **task** and **process** is used inter-changeably. Conceptually, these terms refer to the computational/processing element in a Kahn Process Network.

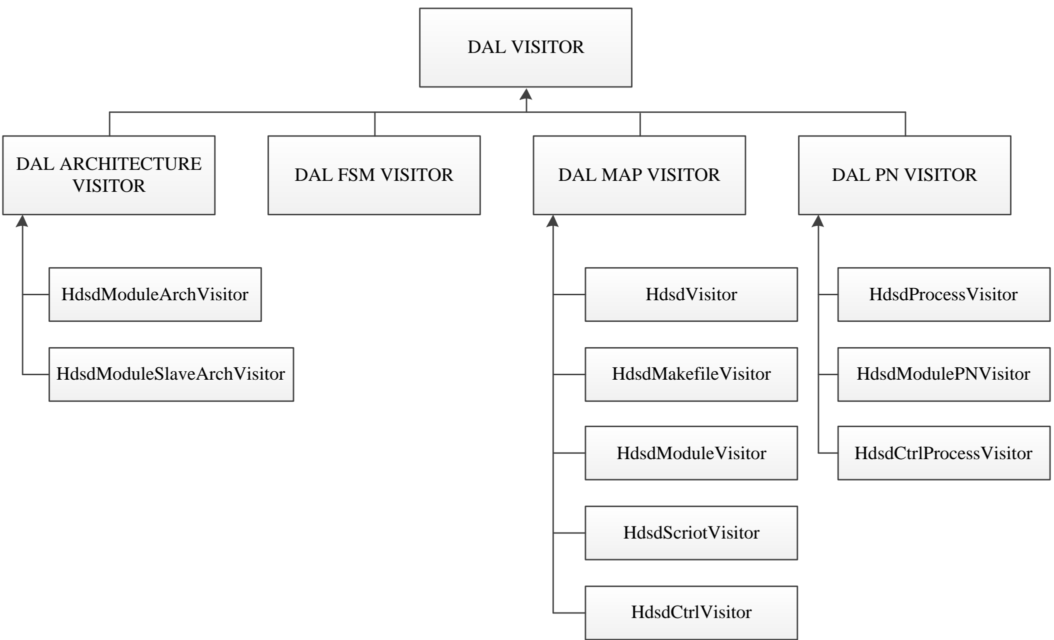


Figure 5.1: DAL Visitor work-flow for SystemC Distributed (hdsd visitor).

5.2 Master Controller

The addition of application control to the SystemC application code is one of the major design code changes. The controller design followed in this work is the Centralized Control Model (Section 3.4.2).

Header File for Master controller

The header file for *master controller* contains all the class based information required for sending the *control messages* to *slave controllers*. The header file begin with the class definition of a master control wrapper. It includes all the port (*sc_port*) declarations. The master controller contains the *initialize()* and *fire()* functions. It also contains master thread control flag namely *_detach* which is used to activate the execution of the master control functionality using *fire()* routine. Since the master has full control over the execution of the system, it sends control messages using the protocol (sub-section 5.4) to slave controllers. The master control wrapper also includes a set of functions for control message creation (per application), sending and receiving. To support run-time task switching between the clone tasks over cloned processor, the master controller has a series of functions to generate the application task-specific control signals. These functions are specific to the *hdsd* based implementation. These functions described in Table 5.1 contains a list of functions that are used to query the Static Table (refer to sub-section 5.3 for more information) entries in master controller to generate the control signal.

In addition to the query functions, the master controller statically generates (based on code generation) a list of task requiring select signals. Each individual task is obtained from the list and functions in Table 5.1 are called for querying. This task is done using functions defined in Table 5.2

The master controller wrapper also contains the declaration of DAL Action functions pertaining to applications defined in the process network[†].

[†]The function defined are not task specific but based on Application Prefix. The Application Prefixes are

Table 5.1: Master Control Query Functions specific to fault tolerance.

Control functions specific to fault tolerance
int **getActiveProcessorIndex**($char * str$) Looks up the *ActiveProcessorTable* based in *str* and return the index.
$char *$ **getActiveProcessorName**($char * str$) Looks up the *ActiveProcessorTable* based in *str* and return the processor name.
void **queryForMuxTask**($char * stateInfo, char * sourceProcessor, char * host_processor, char * task, char * result$) Content matching query to *ApplicationChannelList* for Mux based on State $stateInfo$ of the system, *sourceProcessor* name, *host_processor* name and Application Task ($task$). The *result* stored *task* name with a valid signal.
void **queryForDemuxTask**($char * stateInfo, char * targetProcessor, char * host_processor, char * task, char * result$) Content matching query to *ApplicationChannelList* for Demux based on State $stateInfo$ of the system, *targetProcessor* name, *host_processor* name and Application Task ($task$). The *result* stored *task* name with a valid signal.

Table 5.2: Master Control Functions to generate queries relevant to Tasks and processors.

Query generator control functions
handlerForMuxTaskList($char * stateInfo, char * sourceProcessor, char * host_processor, char * taskList, char * result$) Handles the Mux select signal generation for a List of Task $taskList$
handlerForDemuxTaskList($char * stateInfo, char * targetProcessor, char * host_processor, char * taskList, char * result$) Handles the Demux select signal generation for a List of Task $taskList$
handlerForSourceProcessorList($char * stateInfo, char * sourceProcessorList, char * host_processor, char * taskList, char * result, char * muxCloneInfo$) Handler for managing the Mux signal value *result* and $taskList$ for a list of task distributed over the source processors $sourceProcessorList$.
handlerForTargetProcessorList($char * stateInfo, char * targetProcessorList, char * host_processor, char * taskList, char * result, char * demuxCloneInfo$) Handler for managing the Demux signal value *result* and $taskList$ for a list of task distributed over the target processors $targetProcessorList$.

CPP File for Master Controller

The cpp file for master controller contains all the functional behavior of class functions and helper functions for control. Due to the inclusion of DOL wrapper functions like initialize() and fire() routines, the master controller is treated as a SystemC thread that can be attached to the event scheduler just like an ordinary application task. Based on SystemC scheduling and time-quanta allocated to the master controller, the *fire()* module in wrapper instance is called. The fire routine is built over the FSM description of State Transition Action model. Based on State-Transitions and corresponding Actions List, the master controller fire calls corresponding DAL Action routines namely the following *DAL_Action START*.

- *DAL_ACTION_[App_Prefix]_START*: This function calls for initialization of all application process-specific variables and resets the *_detach* flag. In other words, when this START action is called for an application, the execution begins from scratch.
- *DAL_ACTION_[App_Prefix]_STOP*: When STOP action is called for an application in FSM, the master controller instructs all the slave to call finish() routine for application task. This resets the local data of application task.
- *DAL_ACTION_[App_Prefix]_PAUSE*: If a user programmer wants to preserve the local state of the application (all processes in the application) and then reuse the data in future, PAUSE action can be used. This sets the activity flag *_detach*.
- *DAL_ACTION_[App_Prefix]_RESUME*: The RESUME action can be used to resume the application execution from the last paused state.

These application specific DAL Action functions are created based on the FSM action specification. If a user avoids the usage of PAUSE and RESUME functions in the FSM specification, these functions are not created during code-generation phase. During the code-generation phase, the hdsd-based visitor function provides values for computing visitor-specific protocol fields. The master controller uses the functions mentioned in Table 5.2 with these statically generated information coupled with run-time data (subsection 5.3) to generate the *control message* fields like multiplexer task Informations at run-time. Before sending the *control message*, the master controller checks for the Active destination slave controller (example in Table 5.3) and obtains the port information from Processor Port Table (example Table 5.4). Refer to sub-section 5.3 for more details on the Tables.

Just like an application task wrapper, the master controller also has a local implementation of *DOL_read()* and *DOL_write()* function (same as in DOL middle-ware). On the other hand, the master controller have null definition of other DAL functions like finish, save, etc.

The Figure 5.2 provides a flow diagram of the master controller *fire()* routine. In the figure there are multiple blocks named as application control Functions. These block correspond to functions enlisted above.

5.3 Run-time Table for Application Control

The master controller statically creates look-up tables for creating control decisions at run-time for fault-tolerance simulations. The main look-up tables created are mentioned below:

- ActiveProcessorTable: This table contains a list of entries corresponding to slave processor name along with clones in an indexed fashion. Each processor name is accompanied by it activity flag. The activity flag is governed by the relation 5.1 such that at any given time only one of slaves is active.

$$active_flag \wedge clone_active_flag \tag{5.1}$$

derived from the order for occurrence of *App_Prefix* in the FSM state-transition-action database.

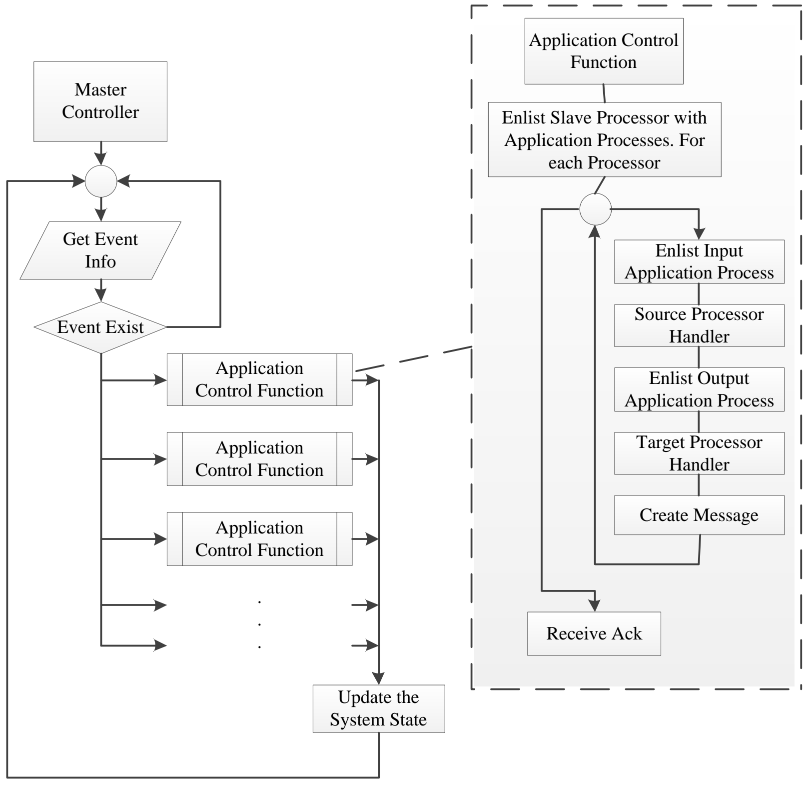


Figure 5.2: Master controller work-flow for HDSD visitor.

Table 5.3: Active Processor Table

Index	Processor		Flag	
	Original	Clone	Original	Clone
0	$sim1$	$sim1_clone$	$TRUE$	$FALSE$
1	$sim2$	$sim2_clone$	$TRUE$	$FALSE$

- MasterProcessorPortTable: The master processor port table consist of an array of port information and its connectivity details to the corresponding slave controlled processor.
- ApplicationChannelTable: The channel name and the connected application tasks is statically stored in the Table called ApplicationChannelTable/ApplicationChannelList. The table is structured in two parts. The first part contains information about to *origin process* and second part deals with *target process*. The origin process contains process name, processor Name, Active State for the given process name and a flag. The flag in either origin and target process Information points to the existence of clone processor towards the target and origin side respectively.

Table 5.4: Master Processor Port Table

Index	Processor	Port Pointer	
		Input	Output
0	*sim1*	*port_IN*	*port_OUT*
1	*sim2*	*port_IN*	*port_OUT*
2	*sim1_clone*	*port_IN*	*port_OUT*
3	*sim2_clone*	*port_IN*	*port_OUT*

The example of ActiveProcessorTable (Table 5.3), MasterPortList (Table 5.4) and ApplicationChannelTable[‡] (Table 5.5) list the entries statically populated to the corresponding tables at compile time. The table contents correspond to the application scenario shown in Appendix A.2.1.

5.4 Control Message

The *control message* carries information pertaining to application control. This message is either created by the master or slave controllers. The slave controller creates control messages to acknowledge the *control message* received from the master. The acknowledgement message sent to master after the slave controller has processed the control message. On the other hand, the master controller creates the Message with control informations like DAL_Action, Application Prefix and other informations (implementation-specific) required for controlling the application at run-time. The master controller uses the DAL database to compute and populate the fields in the *control message*. The generic mandatory fields in *control message* are:

- Flag: Numerical Value to denote the DAL Action.
  - START ⇔ 0x1000
  - STOP ⇔ 0x0100
  - PAUSE ⇔ 0x0010
  - RESUME ⇔ 0x0001

[‡]Due to representation constrains, the names shown in the Table have been shortened. The actual names are shown below. The symbol ⇒ is used to associate an acronym to the actual full name.

- A1 ⇒ APP1 and A2 ⇒ APP2.
- gen ⇒ generator.
- sq ⇒ square.
- con ⇒ consumer.
- St ⇒ State.
- 0 ⇒ boolean FALSE.
- 1 ⇒ boolean TRUE.

Table 5.5: Application Channel Table.

Channel	Origin					Target				
	Process	μP	Port	State	Flag	Process	μP	Port	State	Flag
A1_C1	A1_gen	sim1	out	St1_St4	0	A1_sq	sim2	in	St1_St4	0
A1_C2	A1_sq	sim2	out	St1_St4	0	A1_con	sim2	in	St1_St4	0
A2_C1	A2_gen	sim1	out	St2_St4	0	A2_sq	sim1	in	St2_St4	0
A2_C2	A2_sq	sim1	out	St2_St4	0	A2_con	sim2	in	St2_St4	0
tmp_A1_C1	A1_gen	sim1	out	St1_St4	1	Clone_A1_sq	sim2_clone	in	St1_St4	0
tmp_Clone_A1_C1	Clone_A1_gen	sim1_clone	out	St1_St4	0	A1_sq	sim2	in	St1_St4	1
Clone_A1_C1	Clone_A1_gen	sim1_clone	out	St1_St4	1	Clone_A1_sq	sim2_clone	in	St1_St4	1
tmp_A1_C2	A1_sq	sim2	out	St1_St4	1	Clone_A1_con	sim2_clone	in	St1_St4	0
tmp_Clone_A1_C2	Clone_A1_sq	sim2_clone	out	St1_St4	0	A1_con	sim2	in	St1_St4	1
Clone_A1_C2	Clone_A1_sq	sim2_clone	out	St1_St4	1	Clone_A1_con	sim2_clone	in	St1_St4	1
tmp_A2_C1	A2_gen	sim1	out	St2_St4	1	Clone_A2_sq	sim1_clone	in	St2_St4	0
tmp_Clone_A2_C1	Clone_A2_gen	sim1_clone	out	St2_St4	0	A2_sq	sim1	in	St2_St4	1
Clone_A2_C1	Clone_A2_gen	sim1_clone	out	St2_St4	1	Clone_A2_sq	sim1_clone	in	St2_St4	1
tmp_A2_C2	A2_sq	sim1	out	St2_St4	1	Clone_A2_con	sim2_clone	in	St2_St4	0
tmp_Clone_A2_C2	Clone_A2_sq	sim1_clone	out	St2_St4	0	A2_con	sim2	in	St2_St4	1
Clone_A2_C2	Clone_A2_sq	sim1_clone	out	St2_St4	1	Clone_A2_con	sim2_clone	in	St2_St4	1

- Application Prefix : String value pointing to the application for which the Control message is constructed.

Based in the Control Mode and Visitor, the *control message* may vary in structure and fields. For Centralized Control and Visitor as distributed SystemC (hdsd), the *control message* contains additional fields. These fields may or may not be required for other Visitor implementations.

- Current State: It contains the current State of the entire System. It is represented as $Current_State$ in the packet.
- Next State: It is represented as $Next_State$ and denotes the next transited State of the system.
- Multiplexer Clone Information: This field contains the string information required for constructing the multiplexer select signal for application task; represented as Mux_Clone_Info.
- De-multiplexer Clone Information: The protocol field $Demux_Clone_Info$ contain the element required to construct the de-multiplexer select signal.
- Multiplexer Task List: The element Mux_Task_List lists all the application task for which the field Mux_Clone_Info is required for multiplexer signal.
- De-multiplexer Task List: It contains the name of application tasks for which the $Demux_Clone_Info$ is used to compute. This is used to generate de-multiplexer select signal. In the control message this field is referred to as $Demux_Task_List$.

5.5 Slave Controller

The slave controller is constructed as an application task. The only exception is that, the slave wrapper has an additional class function, $ack()$, along with $initialize()$ and $fire()$ routines. The slave controller receives the *control message* from the master controller. The received message is shared with the SystemC application scope (*scd_sim<number>.cpp*). The received (shared) message is processed in the SystemC application scope. The individual message fields are parsed. Based on the parsed fields activity flags pertaining to the application tasks are updated. The parse fields from the message is also used to compute the select signal and assigned to the respective application task port-based multiplexer/de-multiplexer at run-time. The slave controller sends an acknowledgement message to the master controller after it has processed all the fields of the control message (function: *control_msg_handler()*). The master controller processes the next event/transition_id after it has successfully received the acknowledgement message from the slave controller.

In nutshell, the slave controller is just a dummy thread running on slave processors that receives the Control message from master and dispatches application task signals in the local processor scope.

5.6 Application Task Wrapper

The application task wrapper class is constructed with the Basename as process in a process network. If a process is replicated using *iterator* keyword in processNetwork.xml file, only one process wrapper is constructed; since the Basename is the same for all iterated processes. A list of port-based multiplexer and de-multiplexer is constructed to support both State-based task Migration and task Cloning for fault tolerance. A reference to these multiplexers and de-multiplexers are added to the task/process wrapper. The code-generator also add a series of task

functions. These functions are mentioned below. Amongst these, the functions *setDetached* and *unsetDetached* are used to control the activity of application tasks.

- *initialize*(): The SystemC application calls this routine to initialize all the initial values. Once initialized the task is ready for execution.
- *fire*(): The *fire* routine act as a function encapsulating the application task (C code) implementation of fire routine. The application task contain the computational and communicational aspect of a task. Based on application design there can be any combination of computation and communication.
- *setDetached*(): This function is used to control the task flag *_detach*. It disables the execution of the application task.
- *isDetached*(): The function *isDetached* is used to check state of the task flag *_detach* (set or reset).
- *unsetDetached*(): The task instance can call this function to reset the task flag. After the flag is reset, the task can begin execution.
- *finish*(): The wrapper can clean-up the memory space occupied by the application task by calling the *finish* routine. The wrapper can reset all the process wrapper information. This function is called in response to the occurrence of *STOP* in *DAL_ Action*.
- *save*(): This function is used store the application task context.
- *restore*(): During task Migration, the transition can be invisible to application programmer because the context of the process can be stored to a memory and later replaced with the existing context.

The application task wrapper also contains a local definition of *DOL_read*() and *DOL_write*(). These functions have additional operation unlike the functional definition in DOL Framework. As mentioned earlier, the application task wrapper has a reference to the port based multiplexers and de-multiplexers. The data received from the either channels is read by the multiplexer first and then written back to the application task scope. This is done using a special function that reads data from a given the input multiplexer port using a multiplexer Select signal and write the data to output port of the multiplexer. The output port of the multiplexer is connected to the application wrapper. To distinguish between the multiple input/output port in a wrapper scope, the task wrapper maintains a list of ports (reference to ports). The *DOL_read*/*DOL_write* functions uses the port reference from this list of ports to call the read and write function for the multiplexer and de-multiplexer for a given port. This is required because the *DOL_read*/*DOL_write* with port information is called in application task scope (C code). This information is not known in the wrapper scope.

5.7 SystemC Application

The SystemC application begin with a class definition of *sc_ application*; inheriting the features of SystemC module class (*sc_ module*). The *sc_ application* encapsulates the following information for normal execution.

- Local instance of all multiplexer and de-multiplexer pertaining to application task in SystemC application scope.
- Global definition of select signal for each instance of multiplexer and de-multiplexer.
- Local instance of application task wrappers.

- SystemC event associated to each application task instance.
- Local and remote FIFO channels.
- Temporary local FIFO channels (with prefix *tmp_ch*) for connecting multiplexer and de-multiplexer to the application task wrapper port.

The *sc_application* constructor creates and initializes all the instance of FIFOs and allocates appropriate size (based on specification in *processNetwork.xml*). Post the initialization phase of FIFO and task wrapper instances, the binding information between the *channel port* and *task Mux/De-mux port* and vice-versa are defined. The port binding information can be defined for either of the following categories:

- Binding between task wrapper to channel and vice versa (non-cloned and non-migrated).
- Binding between task wrapper to temporary channel (*tmp_ch*) and vice versa (cloned and/or migrated).
- Binding between multiplexer/de-multiplexer and channel and vice versa (non-cloned and non-migrated).
- Binding between multiplexer/de-multiplexer and channel and vice versa (cloned and/or migrated).
- Binding between multiplexer/de-multiplexer and temporary channel and vice versa (cloned and/or migrated).

As explained in previous section, each wrapper instance contain pointers to port-based multiplexers/de-multiplexers. These pointers are initialized to its respective instance of multiplexer/de-multiplexer instance within the scope of *sc_application*. The select signal is computed and assigned to corresponding multiplexer/de-multiplexer instance at run-time. These values are then used in the task wrapper scope during functional calls to *DOL_read*/*DOL_write*, Figure 5.3.

In addition to the application task specific instance creation and initialization, the *sc_application* initiated creation and initialization for controller slave residing in the local scope. For each application task instance created in the scope, the *sc_application* creates a SystemC thread (*SC_THREAD*). Within the scope of *sc_application*, two special SystemC threads are also created. The first thread is named *thread_init*. This thread is called only once and is used to call the initialize function for all application task wrappers. The second thread is the scheduler thread (*thread_sched*). The scheduler thread processes the Event List and schedules the task/process wrapper threads.

The slave controller thread when scheduled calls the fire() routine to receive the *control message* from the master controller. The slave controller thread calls a routine, *control_msg_handler*, (defined in the scope of *sc_application*) to parse the protocol fields and process the *control message*. The slave controller thread takes appropriate actions (described below) based in the message fields. All these actions are governed by the

- Enable/disable a task by setting/unsetting the value of *_detach* flag.
- Construction of select signal using *next state* of the system and/or multiplexer/de-multiplexer clone information for task-specific multiplexer and/or de-multiplexer.
- Appropriate task wrapper (extracted from *MuxTaskList*/*DemuxTaskList*) function calls based on *DAL_Action*.
- Addition/removal of SystemC events (task-specific) to the EventList and notifying the scheduler thread based on *DAL_Action*.

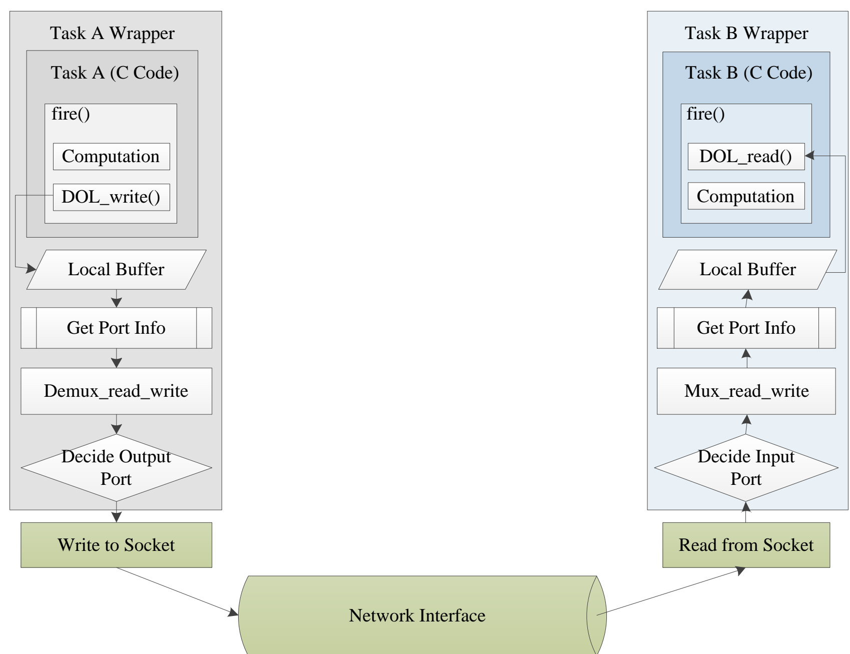


Figure 5.3: Modified *DOL_read()* and *DOL_write()* functions for DAL hdsd-visitor implementation.

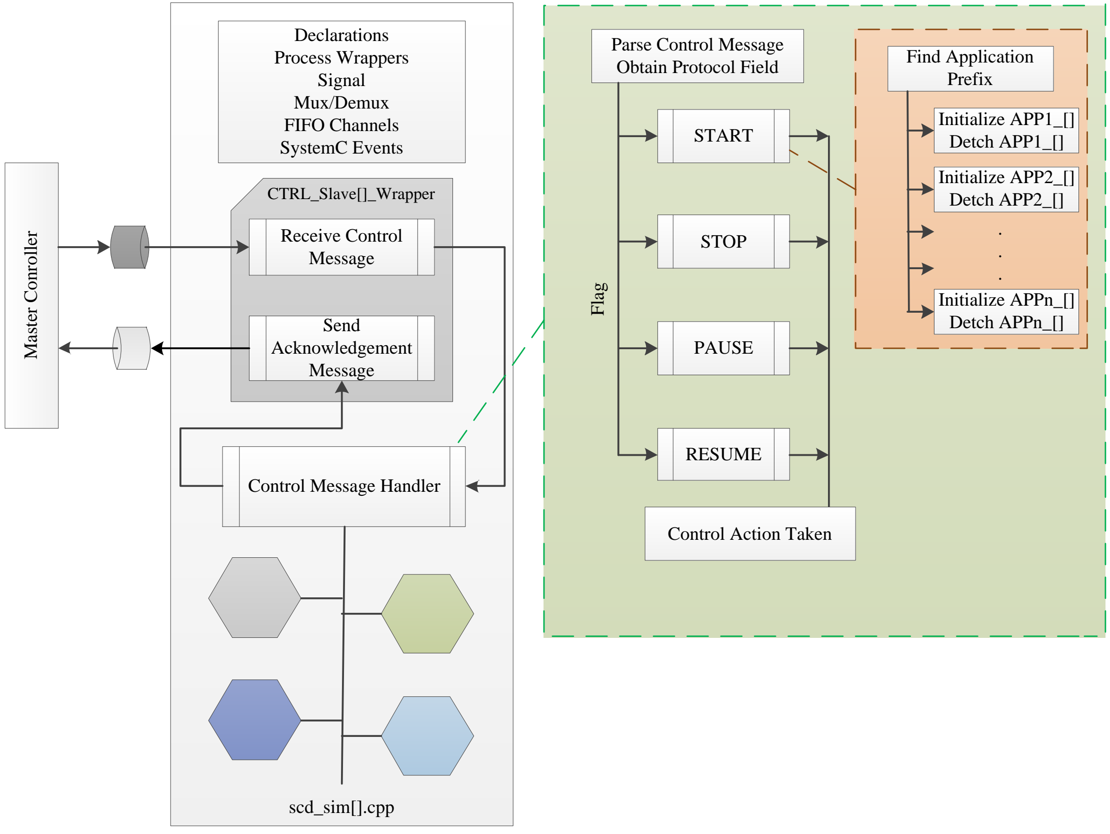


Figure 5.4: The Control Message Processing at the SystemC Application with a Slave controller and few Application processes.

The Figure 5.4 shows the abstract level flow diagram of a SystemC application with a slave controller receiving a message from master controller.

The master controller resides also within the scope of sc_application (file *scd_sim0.cpp* in this implementation). The process wrapper instance creation/initialization, FIFO channel creation and port binding are performed in the same way as other *sc_applications*. But unlike others, this *sc_application* does not encapsulate any thread corresponding to an application task/process wrapper. The encapsulated threads are described below:

- thread_init: The function of this thread is to call the initialize routine of the master controller wrapper.
- thread_sched: This thread is responsible for the scheduling the *thread_CTRL_Master* based in events present in the Event List.
- thread_CTRL_Master: This master controller thread calls the fire routine of the master controller wrapper.

6
Experimental Result

This chapter deals with the case scenarios and experiments performed for a variety of specification. The chapter is broadly classified in two section. The first section deals with simple process networks like *producer-actor-consumer*. The next section deals with the complex process networks in addition to inclusion of simple process networks.

6.1 Simple Applications

The reason behind a simple application scenario is to observe the DAL framework behavior in detail. This can be used as a basis to include complex and real-world application in the example scenarios. The simple application scenarios are shown in the Table 6.1. The Table contains a set of application scenarios. The statistics shown in Table 6.1 contain information about the processes, channel and connections for application scenarios without State-based task migration.

Table 6.1: Simple Process Networks for DAL middle-ware

Sl. No.	App_Prefix	Processes/Tasks	Channels	Connections
1	APP1	Generator Square Consumer	C1 C2	g-ch; ch-s s-ch; ch-c
	APP2	Generator Square2 Consumer	C1 C2	g-ch; ch-s s-ch; ch-c

In case of application information with FSM state-base remapping/migration, the details are shown in Table 6.2. The user can specify a wide range of mapping specification with FSM state-based task migration. The example shown here is the case scenario with all the processes of application *APP1* is mapped with different configuration or migrated to a different processor based on FSM state. The mapping file generated (either manual creation or based in performance analysis and DSE), can have various variants of task migrations. The experiment was carried out for a large sub-set of the cases. The result shown here is the worst case scenario in which all the processes in an application are migrated to different cores based in active states of the application.

The Figure 6.1 shows the variation in the FSM state-transition processing time for the example

Table 6.2: Simple Process Networks for DAL middle-ware with Migrated Tasks

Sl. No.	App_Prefix	Processes/Tasks	Channels	Connections
1	APP1	Generator Square Consumer Generator_Mig	C1 C2 C1_Mig	g-ch; ch-s s-ch; ch-c g-ch_Mig; ch-s_Mig
	APP2	Generator Square2 Consumer Square2_Mig	C1 C2 C1_Mig C2_Mig	g-ch; ch-s s-ch; ch-c g-ch_Mig; ch-s_Mig s-ch_Mig; ch-c_Mig

scenarios in Table 6.1 and from Table 6.2. The dashed horizontal lines in the plot correspond to the average transition time of each example scenarios.

6.2 Complex Applications

This section enlists a combination of complex applications like FFT based application, application with layers of Filters, MPEG-2 decoder and NoC Simulator along with simple application as mentioned in Section 6.1.

Table 6.3: Scenario and corresponding Application Name

Example	Application Name
1	Producer-Square-Consumer; Producer-Multiplier-Consumer
2	Producer-Square-Consumer; Producer-Multiplier-Consumer; NoC Simulator
3	Producer-Square-Consumer; Producer-Multiplier-Consumer; FFT
4	Producer-Square-Consumer; Producer-Multiplier-Consumer; FFT; Filter Application
5	Producer-Square-Consumer; Producer-Multiplier-Consumer; FFT; Filter Application; MPEG
6	Producer-Square-Consumer; Producer-Multiplier-Consumer; FFT; Filter Application; MPEG; NoC Simulator

The Table 6.4 lists the process name, connections and channels specified in the process network file. Using this information, the DAL database is created and updated with information related to cloned and migrated Task. The data shown in Table 6.5 contains all the numerical values of the processes, channels and connections created for successful simulations. The statistics mentioned in the Table does not contain the information about the local FIFO channels created to bind the multiplexer or de-multiplexer to the process wrapper. The application names are mentioned in Table 6.3. These applications are adopted from DOL Framework and examples associated with DOL. The details of the application can be obtained from [53, 54]. The details of semantics used for defining the XML files related to process network, architecture and mapping can be obtained from the report [55].

6.3 FSM State Transition Time

The FSM state transition time is the time taken by the system to process a set of action pertaining to a system transition from current FSM state to next FSM state. The transition time t_{Tr} can be mathematically computed using Equation 6.1.

Table 6.4: Complex Process Networks for DAL middle-ware

Sl. No.	App_Prefix	Process	Channel	Connection
1	APP1	Generator	C1	g-ch; ch-s
		Square	C2	s-ch; ch-c
		Consumer		
	APP2	Generator	C1	g-ch; ch-s
		Square2	C2	s-ch; ch-c
		Consumer		
	APP3	Generator	ipCh_0	ip_Con_0; L1Con_A_0
			ipCh_1	ip_Con_1; L1Con_A_1
		fft2_0_0	ipCh_2	ip_Con_2; L1Con_B_0
			ipCh_3	ip_Con_3; L1Con_B_1
		fft2_0_1	opCh_0	op_Con_0; lastLCon_A_0
			opCh_1	op_Con_1; lastLCon_A_1
		fft2_1_0	opCh_2	op_Con_2; lastLCon_B_0
			opCh_3	op_Con_3; lastLCon_B_1
		fft2_1_1	bCh_0_0	LkCon_A_1_0; FFTCon_A_0_0
			bCh_0_1	LkCon_B_1_0; FFTCon_A_0_1
		Consumer	bCh_0_2	LkCon_A_1_1; FFTCon_B_0_0
			bCh_0_3	LkCon_B_1_1; FFTCon_B_0_1
2	APP1	Generator	C1	g-ch; ch-s
		Square	C2	s-ch; ch-c
		Consumer		
	APP2	Generator	C1	g-ch; ch-s
		Square2	C2	s-ch; ch-c
		Consumer		
	APP3	Generator	ipCh_0	ip_Con_0; L1Con_A_0
			ipCh_1	ip_Con_1; L1Con_A_1
		fft2_0_0	ipCh_2	ip_Con_2; L1Con_B_0
			ipCh_3	ip_Con_3; L1Con_B_1
		fft2_0_1	opCh_0	op_Con_0; lastLCon_A_0
			opCh_1	op_Con_1; lastLCon_A_1
		fft2_1_0	opCh_2	op_Con_2; lastLCon_B_0
			opCh_3	op_Con_3; lastLCon_B_1
		fft2_1_1	bCh_0_0	LkCon_A_1_0; FFTCon_A_0_0
			bCh_0_1	LkCon_B_1_0; FFTCon_A_0_1
		Consumer	bCh_0_2	LkCon_A_1_1; FFTCon_B_0_0
			bCh_0_3	LkCon_B_1_1; FFTCon_B_0_1
	APP4		ipCh	ipCon; fConIn
		Producer	opCh	opCon; fConOut
		filter_0	fChA_0	fOutA_InA1_0; fOutA_InA2_0
		filter_1	fChA_1	fOutA_InA1_1; fOutA_InA2_1
		filter_2	fChB_0	fOutB_InB1_0; fOutB_InB2_0
		Consumer	fChB_1	fOutB_InB1_1; fOutB_InB2_1
			fChB_1	sConA1; sConA2

Table 6.5: Application Scenarios and related Statistics

Example	States	Processes	Channels	Connections	Processors	Binding
1	5	21	32	64	5	21
2	8	59	120	240	5	59
3	8	33	84	168	5	33
4	12	41	108	216	5	41
5	16	53	124	248	5	53
6	16	97	220	440	5	97

$$t_{Tr} = \sum_{i=1}^{Act_{\mathcal{N}}} [t_{Act_i}^{Pkt} + t_{SndRcv_i}^{Pkt} + t_{parse_i}^{Pkt} + t_{handling_i}^{Pkt}] + \sum_{i=1}^{Act_{\mathcal{N}}} [\hat{t}_i^{Ack} + \hat{t}_{SndRcv_i}^{Ack} + \hat{t}_{process_i}^{Ack}] \tag{6.1}$$

In the Equation 6.1, the value is computed for each instance of FSM action as specified in the FSM specification. Once the transition identifier is obtained from the list of events, the master controller calls an appropriated *DAL_Action_[Application Prefix]_[Action]()* and constructs a control packet. The time take to construct a control packet is $t_{Act_i}^{Pkt}$. This packet is sent to the slave controller using *DOL_write()*. The packet is received by the slave controller. The packet travel time is referred to by $t_{SndRcv_i}^{pkt}$. The packet is then parsed in the SystemC scope and appropriate action is taken based in the packet content ($t_{handling}^{Pkt}$).

The slave controller sends a acknowledgement message to the controller. The slave controller constructs a control message in $\hat{t}_i^{Ack}$. The flight time of the acknowledgement message is denoted by $\hat{t}_{SndRcv_i}^{Ack}$. On reception, the master controller handles the Acknowledgement message ($\hat{t}_{process_i}^{Ack}$) and proceeds to the next event processing.

Other than the above mentioned factors, the Transition time is also governed by

- The CPU load of the host system/sever on which the simulation is running.
- The status of cache/main memory of the host system/sever also plays a major role. As the memory overhead pertaining to each the application process wrapper is quite high, the control action taken at processor/simulator level might lead to cache miss or page fault. This increase the value of $t_{handling_i}^{Pkt}$ considerably.
- The FSM state transition time is also governed by the granularity of application. If the application is fine grained, there are more number of process with lesser computation, hence more time is consumed in creating the control packet, parsing it and processing it. This is evident in Example3 and Example7 in Figure 6.1. The NoC simulator is a fine grained KPN application and hence the transition time peaks higher and average is much higher than other examples (Figure 6.1 and Table 6.6).

Table 6.6: Average Event Processing Time

Example Number	Average State Transition Time (*seconds*)
1	0.0066230
2	0.0074230
3	0.0064343
4	0.0052199
5	0.0052314
6	0.0105790

6.4 FIFO Performance

In DAL the data flow, as seen in Figure 5.3, from the *DAL_write()* to *DAL_read()* is changed considerably with respect to DOL. The end-to-end packet flight time has increased due to run-time DAL checks, port and channel selection procedures (here the end-to-end flight time refer to

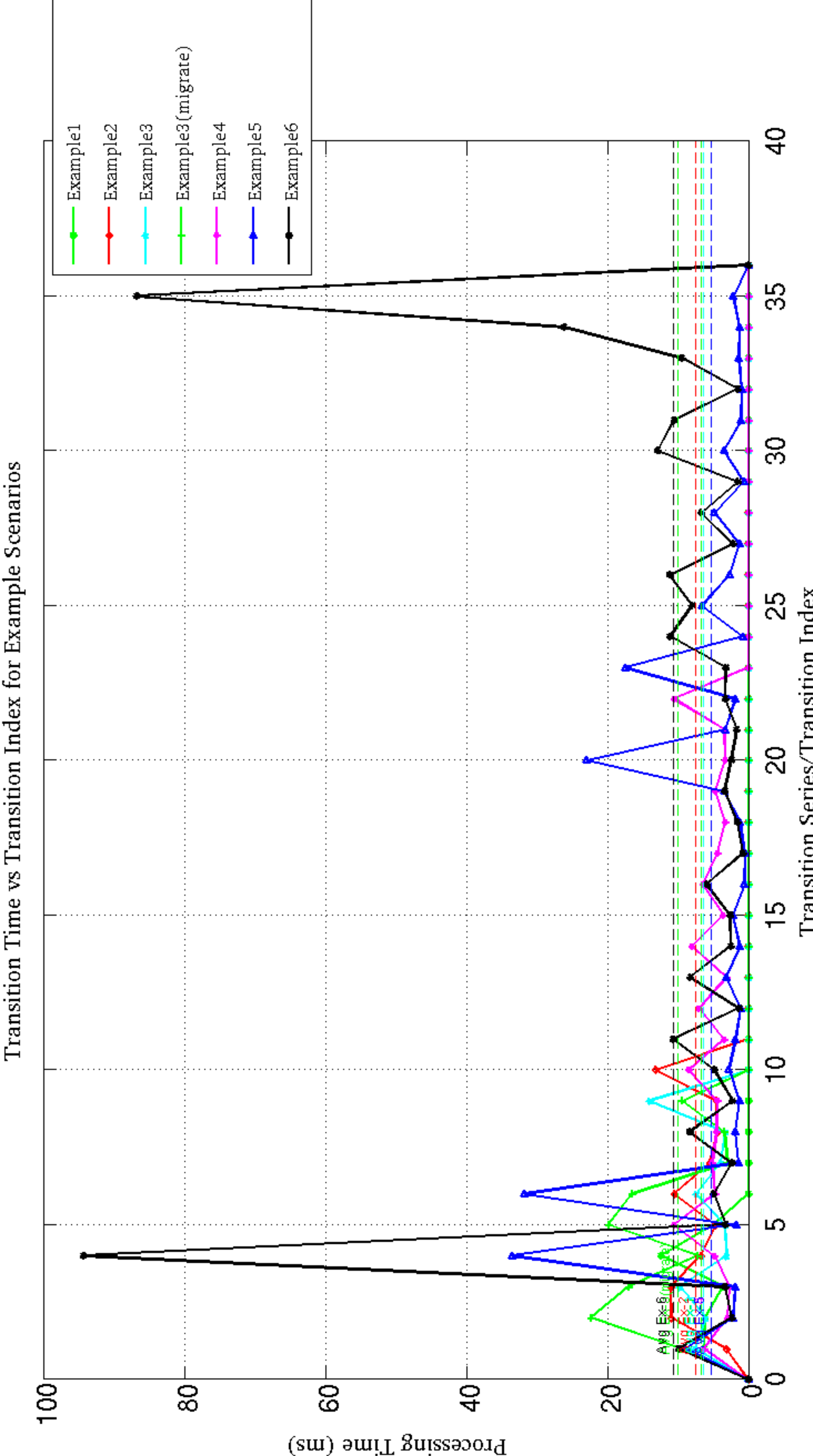


Figure 6.1: State Transition Processing Time for complex application scenarios.

the duration of time taken by source process to write data using *DAL_write()* and the destination process to read the data using *DAL_read()*).

The Table 6.7 shows the average FIFO performance (end-to-end FIFO performance time) of DOL and DAL Framework for both local FIFO and remote FIFO (supported by SCD). As observed, the local FIFO access take four times more in DAL than in DOL Framework. This is due to inclusion of multiplexer and de-multiplexer and related processing time.

Table 6.7: Local and Remote FIFO Performance (Average)

FIFO Type	DOL	DAL
Local FIFO	1.2 μsec	5 μsec
Remote FIFO	300 μsec	450 μsec

The average flight/travel time of a packet over a remote FIFO is approximately half times the value as observed in DOL Framework. The values shown here is the average of the mean value obtained in different conditions. These value may change based on the simulation set-up (like the compile time optimizations) and underlying OS and Hardware configuration. In general the value of packet fight time for remote FIFO is governed by following factors:

- Network load.
- System/server load on the network interface.
- Size of data written and read by the application process.
- Presence of right page in the main memory.

7 Conclusion and Future Works

7.1 Conclusion

This master thesis proposed the specification and implementation of dynamism with respect to multiple co-existing and co-executing applications using a Distributed Application Layer framework for a MPSoC platform namely EURETILE. The initial work started with the creation of a application scenarios i.e. collection of multiple application created with KPN model of computation. The application scenario creation is initiated and managed by the the user programmer. The user programmer also specifies the run-time behavior of the application using a series of events and a finite state machine to express the application dynamism using control APIs like START, STOP, PAUSE or RESUME.

All these specifications are expressed in the XML format and parsed by the DAL framework. The DAL framework also adds a list of controller tasks namely master controller and slave controller to implement event drive application control using the user defined FSM specification. As the framework is specifically designed for a MPSoC platform, a manually generated mapping file is used to geographically distributed application processes onto the different processors*. The DAL framework support state based mapping of applications. This feature is provided such that the presence of multiple applications and event driven system behavior should not be compromised with system performance. The DAL framework also introduced the concept of application based fault tolerance. To incorporate all these specification-related changes, the data-model of the DOL was extended with new fields and elements.

The final simulation environment was developed on top of SystemC with a added library to support distributed execution of SystemC applications (SystemC distributed library). At the code generation level, the selection of static remapping and double redundancy based fault tolerance was governed by the limitation in SystemC standard and implementation. The SystemC standard does not allow dynamic mapping of a *sc_interface* like a FIFO channel to a *sc_port*. Due to this, the channels and processes were duplicated. At run-time, the selection of an appropriate channel for sending and receiving data was managed by the addition of multiplexer and demultiplexer block and a controller generated select signal. Addition of these blocks or layers increase the overall simulation processing time.

*The term processor is used in this thesis work is in accordance to the architectural specification schema used in the framework. The schema supports the concept of specifying a processor as a functional simulation *NETSIM* and used IP address and port address of the server

In term of event driven simulation, the master controller maintained the current state of the system. Based on some external event (manually fed to the master controller using a series of event names), the master initiates the state transition. The master initiates creation of control messages for a given application and sends to all the slave controllers residing on the processor with at least one mapped process related to that application. The master controller waits for an acknowledgement from the slave controller. The slave receives the message and calls the application process wrapper functions and creates an appropriate select signal for multiplexer and demultiplexer. After processing the packet, the slave controller creates and sends a acknowledgement packet to the master controller. The system current state information is then updated.

In the case of emulated fault scenario, the master controller uses the statically generated DAL data model and make control decision on which processor or which application processes to active. All these changes were tested with many applications and varying combination of application (application scenario) at the functional simulation level. Addition of modules to support dynamism has lead to changes in the FIFO accesses like channel selection based in select signal has altered the performance time with respect to DOL framework. Both at local and remote FIFO level, the average FIFO performance has increased. The increase in performance time can be justified with the addition of run-time control of applications and fault tolerance support.

At execution level, in comparisons to the sequential implementation, the KPN based implementation showed considerable speedups. This is subjected to the fact that the computation time is greater than communicational time. In other words, the higher simulation speed-up can also be selected by careful selection of application granularity. Nevertheless, if the computation time is lesser than communication time, considerable amount of data is buffered at the network interface level. As the SCD library do not provide a synchronized remote FIFO implementation, the principle of context saving for task remapping could not be supported.

An important remark is that the framework does not address the task of creating a process network out of a sequential application. This remains the task of the software engineer. improvement This step is very important because if the load of the application is not well shared among all processes, only a small speed-up can be achieved compared to the sequential application.

7.2 Future Works

There are several possibilities to improve and extend the current implementation. To begin with addition of new applications to the execution framework like GSM encoding and decoding, mp3 decoding, etc for better real-world simulation even at functional level. Due to varying application needs, the system performance can be better estimated and analyzed.

The inclusion of synchronized remote FIFO to this functional simulation platform could be done to prevent the unnecessary buffering of data at the network interface level due to varying computational requirements of application processes. This could possibly be achieved by the incorporation of an acknowledge mechanism at the socket layer for every data packet that is forwarded or received using read and write functions. In addition to this, each FIFO created in SystemC should be mapped separately to a new port number. This would prevent race condition due to multiple access as well. The addition of synchronized FIFO would also assist in the context saving process as well. Since the origin process cannot write more data to the FIFO channel, there is no buffering at the interface level and hence, the context of channel can be save without any inconsistent behavior. Using this, periodic check-pointing for process context and FIFO can be supported along with context-based task remapping.

There is still scope of optimization in some piece of implementation. This could be taken into consideration if the code base is used as a basis for generating new run-time environments.

Due to SystemC standard limitation, the functional simulation can also be achieved by developing a distributed simulation using Posix thread and MPI [56, 57]. This set-up could also be used

directly over the multi-core platform from Intel i.e. Intel SCC [58]. The use of Posix thread increases the overall simulation time due to heavy kernel processing during context switching at thread level. On the other hand, if a similar platform is built on a real-time OS like RT Linux or FreeRTOS, the DAL programmer can better exploit the resources like scheduling. Moreover, the memory footprint of the final binaries will be much smaller.

The future works might also include the generation of event for State change by few selected applications. These application can be tagged as privilege and list of event names can be provided to privileged applications. These application can send an event to controller task or can directly add it to a shared on-line list of events.

Appendix

A.1 DOL Specification

This appendix lists a detailed DOL specification of the simple multiprocessor streaming application shown in Fig. A.1 and the results of various design steps.

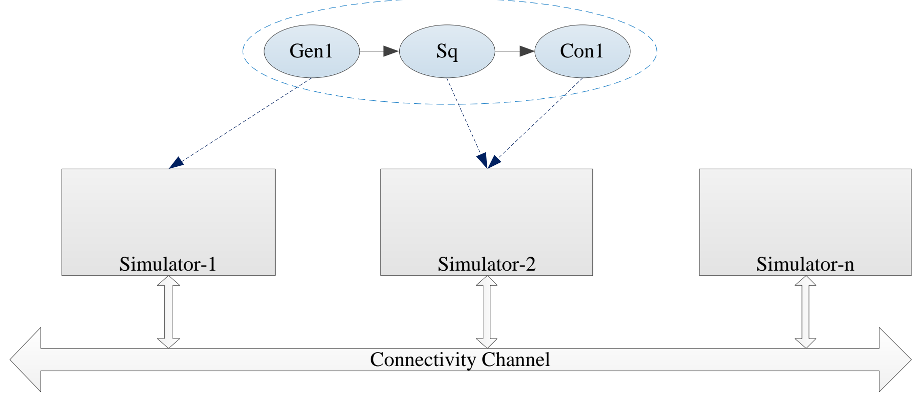


Figure A.1: A simple Process Network and its mapping onto SCD using DOL Framework.

A.1.1 Application and System Specification

This section lists the XML specifications of the process network, the architecture, and the mapping of the system depicted in Fig. A.1. In addition, the C source code of an actor is shown as well as two possibilities to visualize these specifications.

Process Network Specification

The Listing A.1 contains the a simple Process Network Specification of an application containing a Producer, a Consumer and an Actor as a Square. The Specification shown in the Listing is in

XML format. This acts as an input to DOL Framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<processnetwork
    xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK
    http://www.tik.ee.ethz.ch/~shapes/schema/processnetwork.xsd"
    name="producer_comsumer">

  <!-- Processes -->
  <process name="generator">
    <port type="output"   name="out"/>
    <source type="c"  location="generator.c"/>
  </process>
  <process name="consumer">
    <port type="input"    name="in"/>
    <source type="c"  location="consumer.c"/>
  </process>
  <process name="square">
    <port type="input"    name="in"/>
    <port type="output"   name="out"/>
    <source type="c"  location="square.c"/>
  </process>

  <!-- Software Channels -->
  <sw_channel type="fifo" size="10" name="C1">
    <port type="input"    name="in"/>
    <port type="output"   name="out"/>
  </sw_channel>
  <sw_channel type="fifo" size="10" name="C2">
   <port type="input"     name="in"/>
    <port type="output"   name="out"/>
  </sw_channel>

  <!-- Connections -->
  <connection name="g-c">
    <origin name="generator"> <port name="out"/>  </origin>
    <target name="C1">    <port name="in"/>  </target>
  </connection>
  <connection name="c-c">
    <origin name="C2">    <port name="out"/>  </origin>
    <target name="consumer">  <port name="in"/>  </target>
  </connection>
  <connection name="s-c">
    <origin name="square">    <port name="out"/>  </origin>
    <target name="C2">    <port name="in"/>  </target>
  </connection>
  <connection name="c-s">
    <origin name="C1">    <port name="out"/>  </origin>
    <target name="square">    <port name="in"/>  </target>
  </connection>
</processnetwork>
```

Listing A.1: Process Network file specifies the structure of an application. The file includes the process names, software channels and connection informations.

Architectural Specification

The Listing A.2 contains the an Architectural Specification of the SCD simulator. The sim1 acts like a Master and initiates all the Socket based connection with the Slaves (in SCD scope). The Architectural Specification is also is in XML format. This acts as the second input to DOL Framework. This is not created by the user.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture
    xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/ARCHITECTURE"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/
    http://www.tik.ee.ethz.ch/~shapes/schema/architecture.xsd"
    name="architecture">

  <processor name="sim1" type="NETSIM">
    <configuration name="address" value="127.0.0.1" />
    <configuration name="port" value="5877" />
    <configuration name="master" value="true" />
  </processor>
  <processor name="sim2" type="NETSIM">
```

```
15        <configuration name="address" value="127.0.0.1" />
16        <configuration name="port" value="5878" />
17      </processor>
18      <processor name="sim3" type="NETSIM">
19        <configuration name="address" value="127.0.0.1" />
20        <configuration name="port" value="5879" />
21      </processor>
22      <processor name="sim4" type="NETSIM">
23        <configuration name="address" value="127.0.0.1" />
24        <configuration name="port" value="5880" />
25      </processor>
26      <processor name="sim5" type="NETSIM">
27        <configuration name="address" value="127.0.0.1" />
28        <configuration name="port" value="5881" />
29      </processor>
30    </architecture>
```

Listing A.2: Architecture file specifies the structure of underlying hardware. The architecture file mentioned here is specific to SystemC Distributed function simulation.

Mapping Specification

The Listing A.3 contains the Mapping Specification of Application Processes onto SCD simulator. The Mapping file is generated using Y-chart approach (Section 2.2.3). The Mapping Specification is also constructed using DOL Framework in XML format (uniformity in expressing the system characteristics).

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <mapping
3       xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING
6       http://www.tik.ee.ethz.ch/~shapes/schema/mapping.xsd"
7       name="producer_comsumer_mapping">
8
9     <binding name="generator" xsi:type="computation">
10        <process name="generator" />
11        <processor name="sim1" />
12    </binding>
13    <binding name="consumer" xsi:type="computation">
14        <process name="consumer" />
15        <processor name="sim2" />
16    </binding>
17    <binding name="square" xsi:type="computation">
18        <process name="square" />
19        <processor name="sim2" />
20    </binding>
21  </mapping>
```

Listing A.3: Mapping file specifies the mapping of resources onto a given hardware component.

A.2 DAL Specification

This appendix lists a detailed multi-processor applications specification with regards to DOL adaption to EURETILE specification as DAL (as mentioned in Section 2.4. The Fig A.2 shows a simple real world scenario with two active applications on the hardware.

A.2.1 Scenario based Applications and System Specification

This section lists the XML specifications of the FSM , the process network, the architecture (including a master controller processor), and the mapping of the system depicted in Fig. A.2.

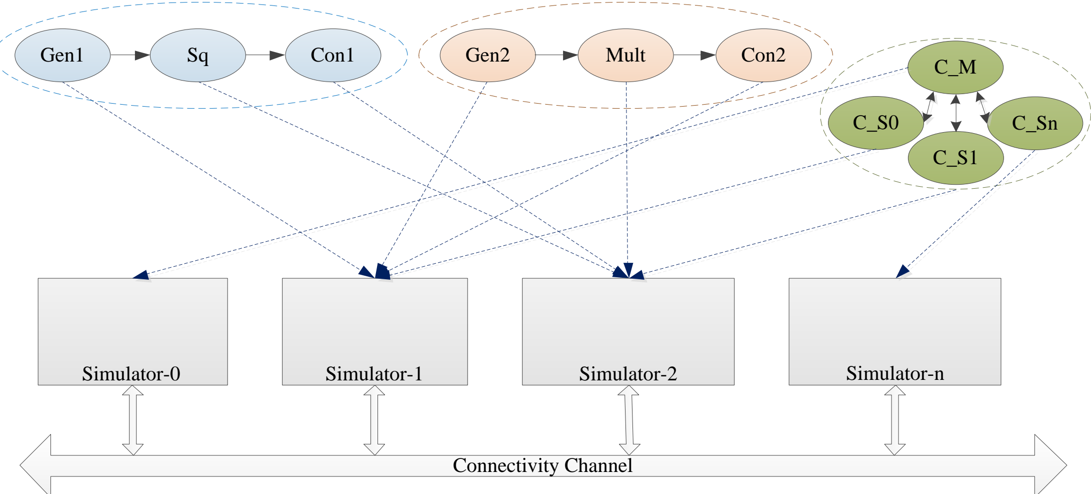


Figure A.2: Simple Application Process Networks and its mapping onto SCD using DAL Framework.

Process Network Specification

The Listing A.4 contains the a Process Network Specification of a scenario consisting of two Applications. The Applications are of the structure a Producer, a Consumer and connected by an Actor in between. The Specification shown in the Listing is generated after merging the Process Network of two Applications namely APP1 and APP2. The merged XML file is one of the four input to DAL Framework.

During execution, the merged Application Process Network is merged with Control Process Network (Section A.4).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <processnetwork
3      xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK
6      http://www.tik.ee.ethz.ch/~shapes/schema/processnetwork.xsd" name="ex1">
7
8  <!-- APPICATION 1 -->
9    <!-- processes -->
10   <process name="APP1_generator">
11     <port type="output"   name="out"/>
12     <source type="c"  location="generator.c"/>
13   </process>
14   <process name="APP1_consumer">
15     <port type="input"    name="in"/>
16     <source type="c"  location="consumer.c"/>
17   </process>
18   <process name="APP1_square">
19     <port type="input"    name="in"/>
20     <port type="output"   name="out"/>
21     <source type="c"  location="square.c"/>
22   </process>
23
24   <!-- sw_channels -->
25   <sw_channel type="fifo" size="10" name="APP1_C1">
26     <port type="input"    name="in"/>
27     <port type="output"   name="out"/>
28   </sw_channel>
29   <sw_channel type="fifo" size="10" name="APP1_C2">
30     <port type="input"    name="in"/>
31     <port type="output"   name="out"/>
32   </sw_channel>
33
34   <!-- connections -->
35   <connection name="APP1_g-c">
36     <origin name="APP1_generator">    <port name="out"/>    </origin>
37     <target name="APP1_C1">       <port name="in"/>    </target>
```

```
    </connection>
    <connection name="APP1_c-c">
      <origin name="APP1_C2">       <port name="out"/>   </origin>
      <target name="APP1_consumer"> <port name="in"/>    </target>
    </connection>
    <connection name="APP1_s-c">
      <origin name="APP1_square">   <port name="out"/>   </origin>
      <target name="APP1_C2">       <port name="in"/>    </target>
    </connection>
    <connection name="APP1_c-s">
      <origin name="APP1_C1">       <port name="out"/>   </origin>
      <target name="APP1_square">   <port name="in"/>    </target>
    </connection>

    <!-- APPLICATION 2 -->
    <!-- processes -->
    <process name="APP2_generator">
      <port type="output"    name="out"/>
      <source type="c" location="generator2.c"/>
    </process>
    <process name="APP2_consumer">
      <port type="input"     name="in"/>
      <source type="c" location="consumer2.c"/>
    </process>
    <process name="APP2_multiplier">
      <port type="input"     name="in"/>
      <port type="output"    name="out"/>
      <source type="c" location="multiplier2.c"/>
    </process>

    <!-- sw_channels -->
    <sw_channel type="fifo" size="10" name="APP2_C1">
      <port type="input"     name="in"/>
      <port type="output"    name="out"/>
    </sw_channel>
    <sw_channel type="fifo" size="10" name="APP2_C2">
      <port type="input"     name="in"/>
      <port type="output"    name="out"/>
    </sw_channel>

    <!-- connections -->
    <connection name="APP2_g-c">
      <origin name="APP2_generator">    <port name="out"/>   </origin>
      <target name="APP2_C1">    <port name="in"/>    </target>
    </connection>
   <connection name="APP2_c-c">
      <origin name="APP2_C2">    <port name="out"/>   </origin>
      <target name="APP2_consumer"> <port name="in"/>    </target>
    </connection>
    <connection name="APP2_s-c">
      <origin name="APP2_multiplier">    <port name="out"/>   </origin>
      <target name="APP2_C2">    <port name="in"/>    </target>
    </connection>
    <connection name="APP2_c-s">
      <origin name="APP2_C1">    <port name="out"/>   </origin>
      <target name="APP2_multiplier">    <port name="in"/>    </target>
    </connection>
  </processnetwork>
```

Listing A.4: Process Network file specifies the structure of two distinct applications. The file includes the application specific process names, software channels and connection informations.

Controller Process Specification

The Listing A.5 specifies the controller processes expressed in the XML format. This pre-created and user need not hand compile this data. Each name tag of controller process, controller channels and connections follow the nomenclature.

```
<process name="CTRL_Master" basename="CTRL_Master">
  <port name="in_0" type="input"/>
  <port name="out_0" type="output"/>
  <port name="in_1" type="input"/>
  <port name="out_1" type="output"/>
  <source location="ctrl_master.c" type="c"/>
</process>

<process name="CTRL_Slave_0" basename="CTRL_Slave_0">
  <port name="in" type="input"/>
```

```
11   <port name="out" type="output"/>
12   <source location="" type="c"/>
13 </process>
14 <process name="CTRL_Slave_1" basename="CTRL_Slave_1">
15   <port name="in" type="input"/>
16   <port name="out" type="output"/>
17   <source location="" type="c"/>
18 </process>
19
20 <sw_channel name="CTRL_Master_Slave_0_0" type="fifo" size="2056">
21   <port name="in" type="input"/>
22   <port name="out" type="output"/>
23 </sw_channel>
24 <sw_channel name="CTRL_Master_Slave_0_1" type="fifo" size="2056">
25   <port name="in" type="input"/>
26   <port name="out" type="output"/>
27 </sw_channel>
28 <sw_channel name="CTRL_Master_Slave_1_0" type="fifo" size="2056">
29   <port name="in" type="input"/>
30   <port name="out" type="output"/>
31 </sw_channel>
32 <sw_channel name="CTRL_Master_Slave_1_1" type="fifo" size="2056">
33   <port name="in" type="input"/>
34   <port name="out" type="output"/>
35 </sw_channel>
36
37 <!-- Master Connections Start -->
38 <connection name="CTRL_Min_0">
39   <origin name="CTRL_Master_Slave_0_0">
40     <port name="out"/>
41   </origin>
42   <target name="CTRL_Master">
43     <port name="in_0"/>
44   </target>
45 </connection>
46 <connection name="CTRL_Mout_0">
47   <origin name="CTRL_Master">
48     <port name="out_0"/>
49   </origin>
50   <target name="CTRL_Master_Slave_0_1">
51     <port name="in"/>
52   </target>
53 </connection>
54 <connection name="CTRL_Min_1">
55   <origin name="CTRL_Master_Slave_1_0">
56     <port name="out"/>
57   </origin>
58   <target name="CTRL_Master">
59     <port name="in_1"/>
60   </target>
61 </connection>
62 <connection name="CTRL_Mout_1">
63   <origin name="CTRL_Master">
64     <port name="out_1"/>
65   </origin>
66   <target name="CTRL_Master_Slave_1_1">
67     <port name="in"/>
68   </target>
69 </connection>
70 <!-- Master Connections End -->
71
72 <!-- Slave Connections Start -->
73 <connection name="CTRL_S_0_in">
74   <origin name="CTRL_Master_Slave_0_1">
75     <port name="out"/>
76   </origin>
77   <target name="CTRL_Slave_0">
78     <port name="in"/>
79   </target>
80 </connection>
81 <connection name="CTRL_S_0_out">
82   <origin name="CTRL_Slave_0">
83     <port name="out"/>
84   </origin>
85   <target name="CTRL_Master_Slave_0_0">
86     <port name="in"/>
87   </target>
88 </connection>
89 <connection name="CTRL_S_1_in">
90   <origin name="CTRL_Master_Slave_1_1">
91     <port name="out"/>
92   </origin>
93   <target name="CTRL_Slave_1">
```

```
94      <port name="in"/>
95    </target>
96  </connection>
97  <connection name="CTRL_S_1_out">
98    <origin name="CTRL_Slave_1">
99      <port name="out"/>
100   </origin>
101   <target name="CTRL_Master_Slave_1_0">
102     <port name="in"/>
103   </target>
104 </connection>
105 <!-- Slave Connections End -->
```

Listing A.5: Controller process specification file. This file is appended to application xml file.

Finite State Machine Specification

The Listing A.6 specifies the Finite State Machine behavior of the Applications for a given set of events stream/transition identifier.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <fsm
3       xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/FSM"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/FSM
6       http://www.tik.ee.ethz.ch/~shapes/schema/fsm.xsd"
7       name="fsm" description="fsm_example1"
8       startstate="Start" currentstate="Start">
9
10    <!-- End States -->
11    <endstate name="Finish"/>
12
13    <!-- States -->
14    <state name="Start">
15      <transition name="tr_start_1" id="tr_start_1" priority="1" newstate="State1" >
16      <action appprefix="APP1" dalaction="START"/>  </transition>
17      <transition name="tr_start_2" id="tr_start_2" priority="1" newstate="State2" >
18      <action appprefix="APP2" dalaction="START"/>  </transition>
19    </state>
20    <state name="State1">
21      <transition name="tr_state1_1" id="tr_state1_1" priority="1" newstate="State2" >
22      <action appprefix="APP1" dalaction="PAUSE"/>
23      <action appprefix="APP2" dalaction="START"/>  </transition>
24      <transition name="tr_state1_2" id="tr_state1_2" priority="1" newstate="State3" >
25      <action appprefix="APP2" dalaction="RESUME"/> </transition>
26    </state>
27    <state name="State2">
28      <transition name="tr_state2_1" id="tr_state2_1" priority="1" newstate="State1" >
29      <action appprefix="APP2" dalaction="PAUSE"/>
30      <action appprefix="APP1" dalaction="START"/>  </transition>
31      <transition name="tr_state2_2" id="tr_state2_2" priority="1" newstate="State3" >
32      <action appprefix="APP1" dalaction="RESUME"/> </transition>
33    </state>
34    <state name="State3">
35      <transition name="tr_state3_1" id="tr_state3_1" priority="1" newstate="Finish" >
36      <action appprefix="APP2" dalaction="STOP"/>
37      <action appprefix="APP1" dalaction="STOP"/>   </transition>
38    </state>
39    <state name="Finish">   </state>
40  </fsm>
```

Listing A.6: FSM file specifies the scenario based activity of the applications residing on the underlying hardware. State transition is triggered by event (one-to-one mapping between events and transitions).

Mapping Specification

The Listing A.7 contains the Mapping Specification of each Application Processes onto series of SCD simulators. The Mapping file is generated using ψ-chart approach (Section 2.4.3). During DAL Framework execution, the Application Mapping Specification is appended with Control Application Mapping information (Listing A.8). This step is invisible to DAL user.

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping
    xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING
    http://www.tik.ee.ethz.ch/~shapes/schema/mapping.xsd" name="ex_mapping">

  <binding name="APP1_generator" xsi:type="computation" state="State1_State3">
    <process name="APP1_generator" />
    <processor name="sim1" />
  </binding>
  <binding name="APP1_consumer" xsi:type="computation" state="State1_State3">
    <process name="APP1_consumer" />
    <processor name="sim2" />
  </binding>
  <binding name="APP1_square" xsi:type="computation" state="State1_State3">
    <process name="APP1_square" />
    <processor name="sim2" />
  </binding>

  <binding name="APP2_consumer" xsi:type="computation" state="State2_State3">
    <process name="APP2_consumer" />
    <processor name="sim1" />
  </binding>
  <binding name="APP2_generator" xsi:type="computation" state="State2_State3">
    <process name="APP2_generator" />
    <processor name="sim2" />
  </binding>
  <binding name="APP2_multiplier" xsi:type="computation" state="State2_State3">
    <process name="APP2_multiplier" />
    <processor name="sim3" />
  </binding>
</mapping>
```

Listing A.7: DAL Mapping file for Application tasks along with the Processor and active State information.

Controller Specification

As the Master and Slave Controllers are also treated as Application Process in DAL Framework, there exist a Controller Process Network and a Controller Mapping Specification. These Specification can be created once statically then reused by the DAL user without the prior knowledge. The Listing A.8 contains the mapping information of the Controller Application.

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping
    xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING
    http://www.tik.ee.ethz.ch/~shapes/schema/mapping.xsd" name="ctrl_mapping">

  <!-- Master Controller in sim0 -->
  <binding name="CTRL_Master" xsi:type="computation">
      <process name="CTRL_Master" />
      <processor name="sim0" />
  </binding>

  <!-- Slave process named as CTRL_slave_1 -->
  <binding name="CTRL_Slave_0" xsi:type="computation">
      <process name="CTRL_Slave_0" />
      <processor name="sim1"/>
  </binding>
  <binding name="CTRL_Slave_1" xsi:type="computation">
      <process name="CTRL_Slave_1" />
      <processor name="sim2"/>
  </binding>
  <binding name="CTRL_Slave_2" xsi:type="computation">
      <process name="CTRL_Slave_2" />
      <processor name="sim3"/>
  </binding>
  <binding name="CTRL_Slave_3" xsi:type="computation">
      <process name="CTRL_Slave_3" />
      <processor name="sim4"/>
  </binding>
  <binding name="CTRL_Slave_4" xsi:type="computation">
      <process name="CTRL_Slave_4" />
      <processor name="sim5"/>
  </binding>
```

```
35 </mapping>
```

Listing A.8: DAL Mapping file for Controllers.

Mapping Specification with State-based Task Migration

A major change in the Mapping Specification in DAL Framework is that an Application Process can be mapped to multiple processor based on mapping optimality. Due to this, each instance of the Application Process with more than one occurrence is identified uniquely. The first mapped instance of an Application can be termed as original process mapping. Forthcoming instance of the application process can be termed as Migrated processes. The Application Process Name and Binding Name can be appended with a string *Migrated[number]*. The Listing A.9 provides a simple example of Mapping Specification for State based Task Migration.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <mapping
3      xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING
6      http://www.tik.ee.ethz.ch/~shapes/schema/mapping.xsd" name="dal_mapping">
7
8    <binding name="APP1_generator" xsi:type="computation" state="State1">
9      <process name="APP1_generator" />
10     <processor name="sim1" />
11   </binding>
12   <binding name="APP1_consumer" xsi:type="computation" state="State1_State3">
13     <process name="APP1_consumer" />
14     <processor name="sim2" />
15   </binding>
16   <binding name="APP1_square" xsi:type="computation" state="State1_State3">
17     <process name="APP1_square" />
18     <processor name="sim2" />
19   </binding>
20
21   <binding name="APP2_consumer" xsi:type="computation" state="State2_State3">
22     <process name="APP2_consumer" />
23     <processor name="sim1" />
24   </binding>
25   <binding name="APP2_generator" xsi:type="computation" state="State2_State3">
26     <process name="APP2_generator" />
27     <processor name="sim2" />
28   </binding>
29   <binding name="APP2_multiplier" xsi:type="computation" state="State2_State3">
30     <process name="APP2_multiplier" />
31     <processor name="sim3" />
32   </binding>
33
34   <binding name="APP1_generator_Migrate0" xsi:type="computation" state="State3">
35     <process name="APP1_generator_Migrate0" />
36     <processor name="sim3" />
37   </binding>
38 </mapping>
```

Listing A.9: DAL Mapping file defining active State of an application task along with the Processor. Based in optimal state based mapping criteria, a migrated task can exist on different cores

A.3 DAL Naming Convention/Nomenclature

This subsection highlights the naming convention followed for database update and code generation for the DAL middle-ware visitor hdsd.

As the DAL is required to support and control execution of multiple application at run-time, each application should be uniquely identified with an Application Prefix. The Application prefix being a string should be kept small. The naming convention followed in this work in context to Application Prefix is **APP[number]**. For instance, the example scenarios shown in Table 6.1 contain a column for Application Prefix.

FSM Specification

The nomenclature followed for FSM is illustrated below:

- State name should not have any special character like underscore. The preferred State name would be similar to *State1*, i.e. a string appended with a number or just a unique string. The string should only be composed of alphabets.
- The finite state machine should begin with a *Start* state and end with a *Finish* state.
- The *appprefix* in Action should contain the Application prefix as per the naming convention mentioned before.
- The DAL action specified should be either of these:
  - START
  - STOP
  - PAUSE
  - RESUME.
- All the alphabets in string assigned to *dalaction* should be capitalized.

Process Network Specification

The structure of Process Network in DAL is same as DOL to allow interoperability. However, few non-required fields have been added to support DAL features on specific visitor. In addition to these elements, the following naming convention is followed for DAL middle-ware.

- The Process *Name* and *Basename* should always be prefixed with a Application Prefix.
- The *Name* and *Basename* corresponding to Channels and Connections should also be prefixed with the Application Prefix.

Architecture Specification

- A Processor can be defined as a string followed by a number.
- In this implementation, the Processor **sim0** is assigned as Master (Section 2.3.2).
- Other Processors *sim[number]* specified in Architecture are Slaves (in SCD terms Section 2.3.2 and Figure 2.5). The value of *number* can be any positive integer greater than zero.
- The Clone Slave Processors are specified using *sim[number]_clone*.

Mapping Specification

The structure of Mapping Specification in DAL contains addition information on top of Map structure as in DOL. The new fields are optional and hence supports interoperability with DOL Framework.

- The computational binding information should be contains name string with an Application Prefix.
- The computational binding includes Active State information of the Process. Multiple active state name should be separated by special character **underscore**. For instance, *State1_State3_State4*.

- The process/task with multiple copies due to State-based Task Migration should be suffixed with the string *_Migrate[number]*. The *number* denoted the number of times the task can be migrated. If there is only one migrated task, the *number* can be omitted from the suffix. The same goes with the binding name as well.
- The clone process/task created to support fault tolerance requirements should be created with name prefixed with a string *Clone[number]_*.
- In case of both Clone and Task Migration, the process name should be illustrated as *Clone[number1]_[App_Prefix]_[Process Name]_Migrate[number2]*.
- If iteration is used to define multiple copies of a process, the task migration for such process (base process) is only possible if all the iterations are migrated.

Controller Specification

- The value assigned to *Name* for Controller Process (either Master or Slave), Channel and Connection should always be begin with **CTRL_**.
- The master Controller is assigned a name *CTRL_Master*. The Slaves are identified by *CTRL_Slave_[number]*. The range of value of *number* is from zero to any higher positive integer value.

New Channel/Connection Specification

The following clauses are applicable to Channel *ch* connecting a source process and a destination process. The $con1, con2$ specify the Connections between source process to channel and channel to destination process respectively.

- The channel and connections are named as *tmp_ch, tmp_con1* and *tmp_con2* respectively, if destination process is a clone of the original process.
- The channel and connections are named as *tmp_Clone[number]_ch, tmp_Clone[number]_con1* and *tmp_Clone[number]_con2* respectively, if source process is a clone of the original process.
- The channel and connections are named as *Clone[number]_ch, Clone[number]_con1* and *Clone[number]_con2* respectively, if both source process and destination process are clones of the original processes.
- The channel and connections are named as *ch_Migrate[number], con1_Migrate[number]* and *con2_Migrate[number]* respectively, if either source process or destination process is required to be migrated to a different processor.
- The channel is named as *tmp_ch_Migrate[number2]* or *tmp_Clone[number1]_ch_Migrate[number2]* or *Clone[number1]_ch_Migrate[number2]* the clone for a migrated process is also required (as per *map.xml*). The same pattern is followed for naming the Connections in this case.

Multiplexer and De-multiplexer

- The Multiplexer is defined per port in the process wrapper. The naming convention followed for multiplexer is *Mux_[Input Port Name]_[Process Name]_wrapper*.
- The naming convention for De-multiplexer followed in this implementation is *De-mux_[Output Port Name]_[Process Name]_wrapper*.

- The input Ports defined in the Multiplexer block are named as *MUX_INPORT_[State Name]*. The output Port is expressed as *MUX_OUTPORT*. In case of fault tolerance support, the Multiplexer block contain the input port for receiving data from clone source. These ports are named as *MUX_INPORT_Clone_[State Name]*.

- The De-multiplexer block includes a series of output Ports named as *DEMUX_OUTPORT_[State Name]*. The input Port receiving data from process wrapper instance is expressed as *DEMUX_INPORT*. Like the Multiplexer block, the De-multiplexer block also contains the output Ports for receiving data from clone sources. These ports are addressed as *DEMUX_OUTPORT_Clone_[State Name]*.

- Each Multiplexer and De-multiplexer is associated with a read-write function. These functions are defined in the process wrapper and are named as *mux_read_write_[Input Port]_[Process Name]_wrapper()* and *demux_read_write_[Output Port]_[Process Name]_wrapper()* respectively.

- The connection between the Multiplexer/De-multiplexer and Process Wrapper instance is handled by a local FIFO named as *tmp_ch_[mux/demux]_[PortName]_for_[ProcessName]*.

Control Message Creation

- The Flag field in control message denote the DAL Actions. In message it is decoded to numeric value. The value $0x1000$ signifies START. STOP is represented by $0x0100$, PAUSE by $0x0010$ and RESUME by $0x0001$.

- The control message contain fields like *mux_task_name* and *demux_task_name* to store the list of task name for which select signal have be constructed at Slave controller. The task names are concatenated to a single string; for instance *APP1_gen$APP1_act1$APP1_act5*. Two task name are separated by a special character **$**.

A.4 Execution Steps

This section deals with the steps followed for execution for a given scenarios.

Step 1:

- Select the applications to be added in the application scenario.

- Copy the source code of each application process in the *source* folder of the *Example/Example[number]* directory.

Step 2:

- Create the Process Network for the each Application and specify using XML.

- Merge all Process Network manually or use the Merge functionality in DAL Framework to merge these XML files into one. Note all the name should be unique especially the name of Iterators and Variables.

Step 3:

- Create the a file containing Architectural Specification as a XML file.
- Check the XML file to contains a processor as a designated *Master*.

Step 4:

- Create the a file containing Architectural Specification (remote simulators) as a XML file.
- Check the XML file to contains a processor as a designated *Master*.

Step 5:

- Create the a file containing Mapping Specification of Applications onto simulators in a XML file.
- Create the XML file in accordance to the Application Specification (specially in accordance to number of iterated processes or channels or connections). Refer to examples created for assistance. The mapping file created in examples is referred to as *map2.xml*.

Step 6:

- Compile the DAL code. The current directory is the DALPATH.
  - ant -f build.xml compile
- The previous version of compiled DAL code can be cleaned using the following command.
  - ant -f build.xml clean

Step 7:

- Execute the Linux Shell script *pre_ compile_ [Example Number].sh* from the DALPATH. Update the pathname used in the *pre_ compile_ [Example Number].sh* script before using it.
- The shell script clears current *pn.xml* and *map_ 2.xml* and replaces it with file without Controller Process Network Information and Mapping Information.

Step 8:

- Change the current working directory to *cd ./build/bin/main*
- Generate the source code of the remote simulators.
  - ant -f runexample.xml -Dnumber=[Example Number]
- This step performs following steps:
  - Merges the User-specified Process Network of Applications with Controller Process Network.
  - Merges the Application Process Mapping with Controller Process Mapping.
  - Validates the specifications and generate DOT files to visualize the Mapping.
  - Generates the source code of the Remote Simulators/Processors as per Process Network, FSM, Architectural and Mapping Specification.

Step 9:

- Change the current working directory using *cd ./example[Example Number]/systemcd/src/*.
- Compile the simulator code using Makefile. Execute *make all* on the console.

Step 10:

- Copy the simulator executable to a directory on its corresponding remote machine (as per *architecture.xml*).
- Open a console on the remote machine and execute the simulator.

Step 11:

- In case recompilation is required for simulator code generation, start from Step 7 A.4.
- In case recompilation is required for DAL, start the execution phase from Step 6 A.4..

B
On-line DAL Example

B.1 DAL Specification

This chapter lists a detailed DAL specification of the two simple KPN application shown in Fig. B.1 mapped onto a 2-core MPSoC simulator.

B.1.1 Application and System Specification

This section lists the XML specifications of the FSM , process networks, architecture, and mapping for example scenario in Figure B.1.

FSM Specification

The Listing B.1 specifies the Finite State Machine behavior of the Applications for a given set of events stream/transition identifier.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <fsm
3    xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/FSM"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/FSM
6      http://www.tik.ee.ethz.ch/~shapes/schema/fsm.xsd" name="fsm"
7    description="fsm_example1"
8    startstate="Start" currentstate="Start">
9
10   <!-- endstates -->
11   <endstate name="Finish"/>
12
13   <state name="Start">
14     <transition name="tr_start_1" nextstate="StateA" >
15     <event name="epsilon"/>
16         <action application="APP1" dal_action="START"/>
17     </transition>
18   </state>
19
20   <state name="StateA">
21     <transition name="tr_stateA_1" nextstate="StateB" >
22     <event name="e_a"/>
23         <action application="APP2" dal_action="START"/>
24     </transition>
25   </state>
26
```

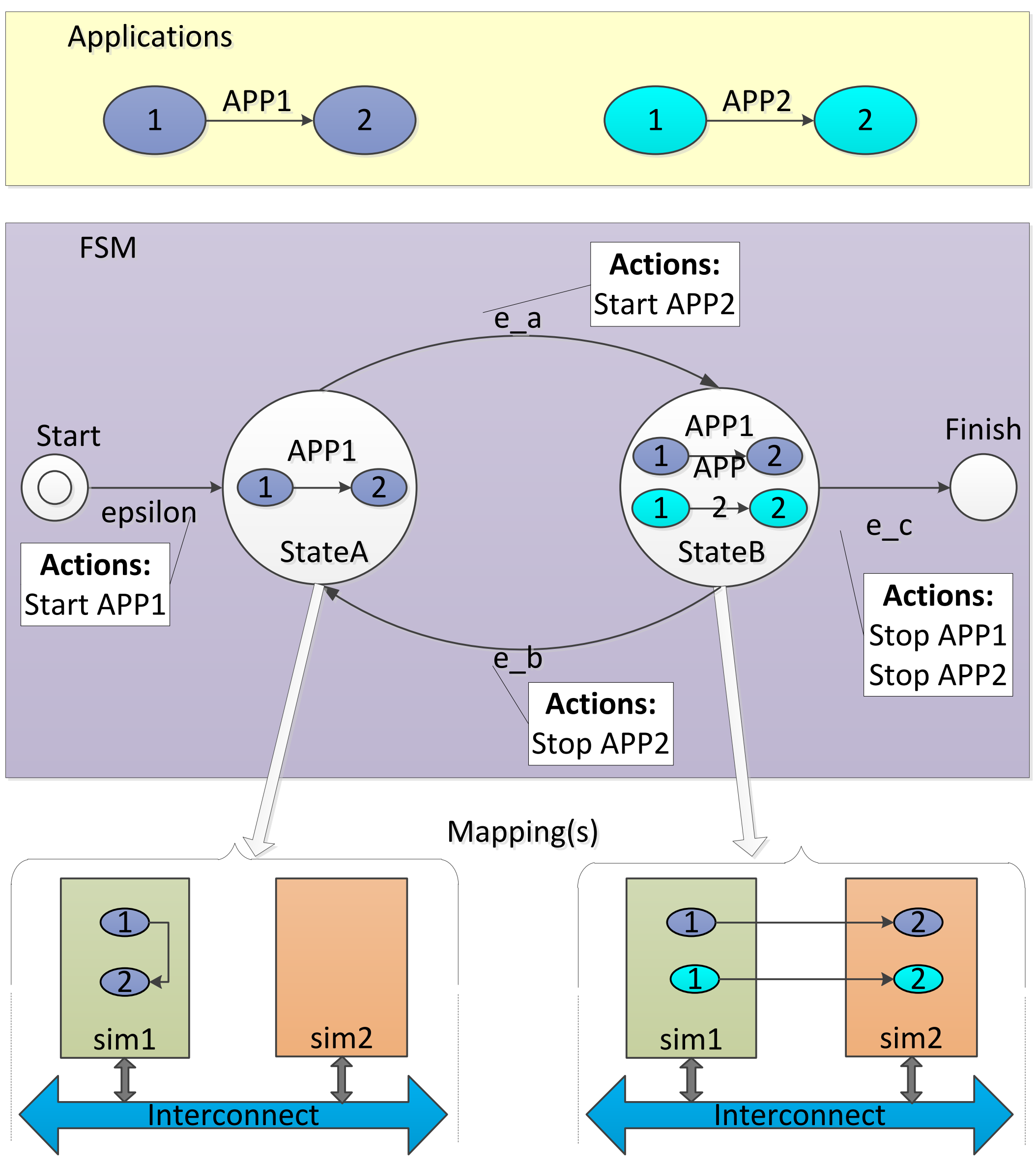


Figure B.1: Online version of DAL along with example specification.

```
  <state name="StateB">
    <transition name="tr_stateB_1" nextstate="StateA" >
    <event name="e_b"/>
        <action application="APP2" dal_action="STOP"/>
    </transition>
    <transition name="tr_stateB_2" nextstate="Finish" >
    <event name="e_c" />
        <action application="APP2" dal_action="STOP"/>
        <action application="APP1" dal_action="STOP"/>
    </transition>
  </state>

  <state name="Finish">
  </state>
</fsm>
```

Listing B.1: FSM.xml.

Process Network Specification

The Listing B.2 contains the a simple process network specification of APP1.

```
  <?xml version="1.0" encoding="UTF-8"?>
<processnetwork
  xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK
    http://www.tik.ee.ethz.ch/~shapes/schema/processnetwork.xsd" name="APP1">

  <!-- processes -->
  <process name="1">
    <port type="output" name="out"/>
     <source type="c" location="generator.c"/>
  </process>
  <process name="2">
    <port type="input" name="in"/>
    <source type="c" location="consumer.c"/>
  </process>

  <!-- sw_channels -->
  <sw_channel type="fifo" size="10" name="C1">
    <port type="input" name="in"/>
    <port type="output" name="out"/>
  </sw_channel>

  <!-- connections -->
  <connection name="g-c">
    <origin name="1">
      <port name="out"/>
    </origin>
    <target name="C1">
      <port name="in"/>
    </target>
  </connection>
  <connection name="c-c">
    <origin name="C1">
      <port name="out"/>
    </origin>
    <target name="2">
      <port name="in"/>
    </target>
  </connection>
</processnetwork>
```

Listing B.2: APP1.xml.

The Listing B.3 contains the a simple process network specification of APP2.

```
  <?xml version="1.0" encoding="UTF-8"?>
<processnetwork
  xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/PROCESSNETWORK
    http://www.tik.ee.ethz.ch/~shapes/schema/processnetwork.xsd" name="APP2">

  <!-- processes -->
  <process name="1">
    <port type="output" name="out"/>
     <source type="c" location="generator2.c"/>
```

```
  </process>
  <process name="2">
    <port type="input" name="in"/>
    <source type="c" location="consumer2.c"/>
  </process>

  <!-- sw_channels -->
  <sw_channel type="fifo" size="10" name="C1">
    <port type="input" name="in"/>
    <port type="output" name="out"/>
  </sw_channel>

  <!-- connections -->
  <connection name="g-c">
    <origin name="1">
      <port name="out"/>
    </origin>
    <target name="C1">
      <port name="in"/>
    </target>
  </connection>
  <connection name="c-c">
    <origin name="C1">
      <port name="out"/>
    </origin>
    <target name="2">
      <port name="in"/>
    </target>
  </connection>
</processnetwork>
```

Listing B.3: APP1.xml.

Architectural Specification

The Listing B.4 contains the an Architectural Specification of the SCD simulator.

```
  <?xml version="1.0" encoding="UTF-8"?>
  <architecture
      xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/ARCHITECTURE"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/
      http://www.tik.ee.ethz.ch/~shapes/schema/architecture.xsd"
      name="architecture">

    <processor name="sim1" type="NETSIM">
      <configuration name="address" value="127.0.0.1" />
      <configuration name="port" value="5877" />
      <configuration name="master" value="true" />
    </processor>
    <processor name="sim2" type="NETSIM">
      <configuration name="address" value="127.0.0.1" />
      <configuration name="port" value="5878" />
    </processor>
    <processor name="sim3" type="NETSIM">
      <configuration name="address" value="127.0.0.1" />
      <configuration name="port" value="5879" />
    </processor>
    <processor name="sim4" type="NETSIM">
      <configuration name="address" value="127.0.0.1" />
      <configuration name="port" value="5880" />
    </processor>
    <processor name="sim5" type="NETSIM">
      <configuration name="address" value="127.0.0.1" />
      <configuration name="port" value="5881" />
    </processor>
  </architecture>
```

Listing B.4: Architecture specification

Mapping Specification

The Listing B.5 contains the Mapping Specification of Application Processes onto MPSoC architecture (SCD simulator).

```
 <?xml version="1.0" encoding="UTF-8"?>
<mapping xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.tik.ee.ethz.ch/~shapes/schema/MAPPING
  http://www.tik.ee.ethz.ch/~shapes/schema/mapping.xsd" name="example0_mapping">

<state name="StateA">
<binding name="APP1_1" xsi:type="computation">
    <process name="APP1_1" />
    <processor name="sim1" />
</binding>

<binding name="APP1_2" xsi:type="computation">
    <process name="APP1_2" />
    <processor name="sim1" />
</binding>
</state>

<state name="StateB">
<binding name="APP1_1" xsi:type="computation">
    <process name="APP1_1" />
    <processor name="sim1" />
</binding>
<binding name="APP1_2" xsi:type="computation">
    <process name="APP1_2" />
    <processor name="s  \begin{subsubsection}{Mapping Specification}im2" />
</binding>

<binding name="APP2_1" xsi:type="computation">
    <process name="APP2_1" />
    <processor name="sim1" />
</binding>
<binding name="APP2_2" xsi:type="computation">
    <process name="APP2_2" />
    <processor name="sim2" />
</binding>
</state>

</mapping>
```

Listing B.5: Mapping file

Events

The Listing B.6 contains the statically generated event stream for FSM state transition.

```
    void dummy_event_gen(int index, char *event)
    {
      //char event[20];
      switch(index)
      {
    case 10:
        strcpy (event ,"epsilon");
        break;
    case 130:
        strcpy (event , "e_a");
        break;
    case 250:
        strcpy (event , "e_b");
        break;
    case 390:
        strcpy (event , "e_a");
        break;
    case 490:
        strcpy (event , "e_b");
        break;
    case 590:
        strcpy (event , "e_a");
        break;
    case 690:
        strcpy (event , "e_c");
        break;
    default:
        event = 0x0;
        break;
      }
    }
```

Listing B.6: Event stream

The final tool chain for code generation and execution is shown in Figure B.2. For more details, please refer to http://www.tik.ee.ethz.ch/ euretile/dal.html.

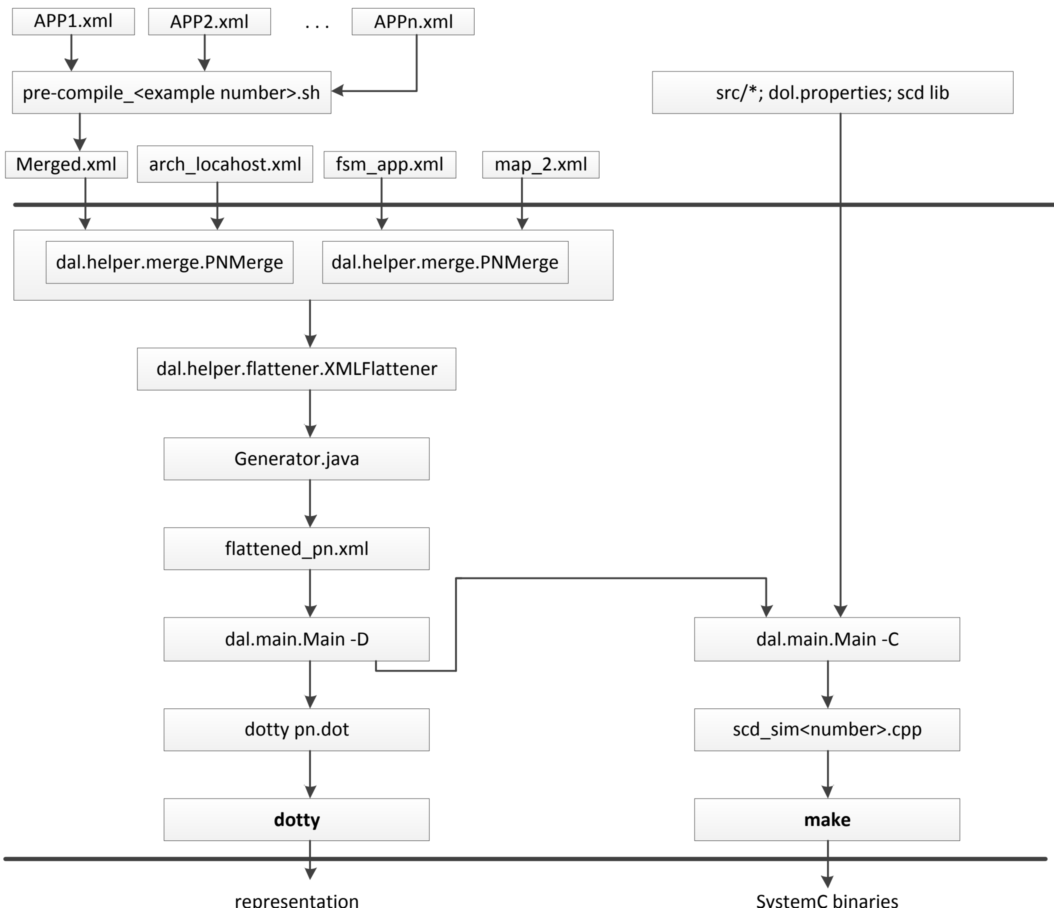


Figure B.2: Tool chain for on-line version of DAL.

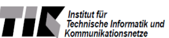
Presentation

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Distributed Simulation of Dynamic and Fault Tolerant System

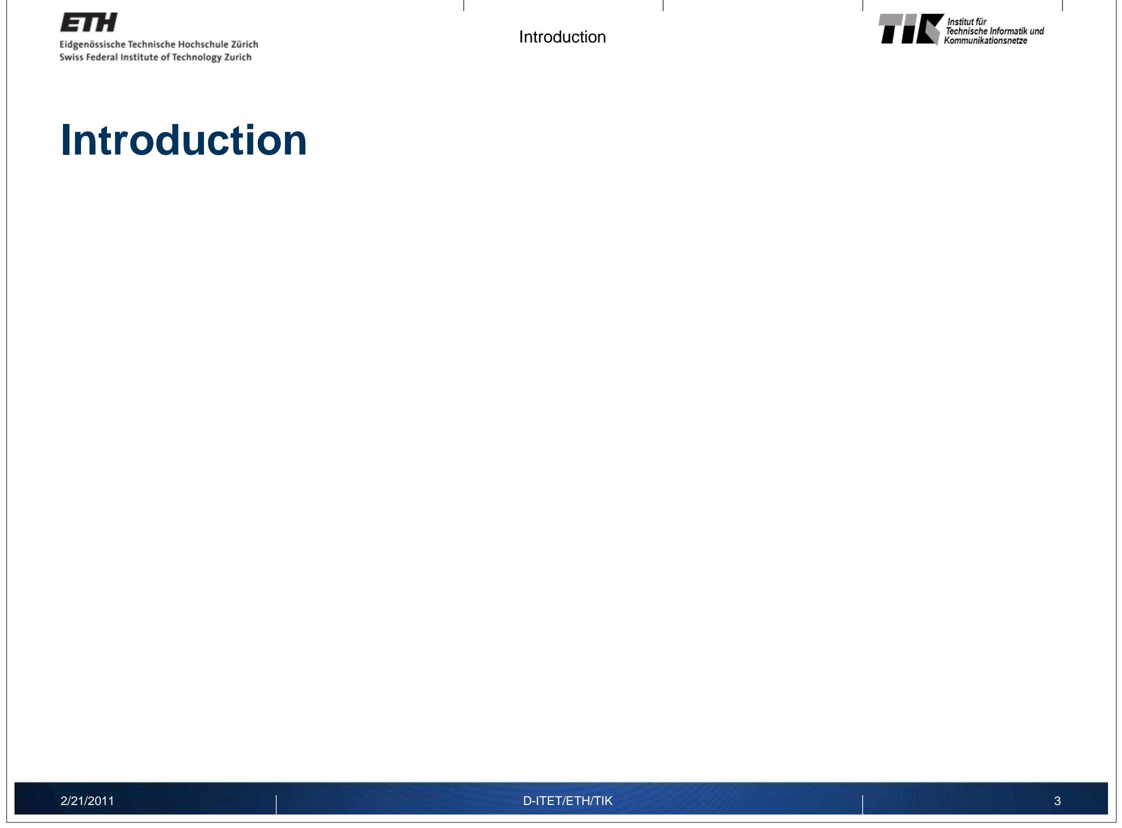
Student: Sudhanshu Shekhar Jha
Advisors: Iuliana Bacivarov, Hoeseok Yang
Professor: Prof. Dr. Lothar Thiele



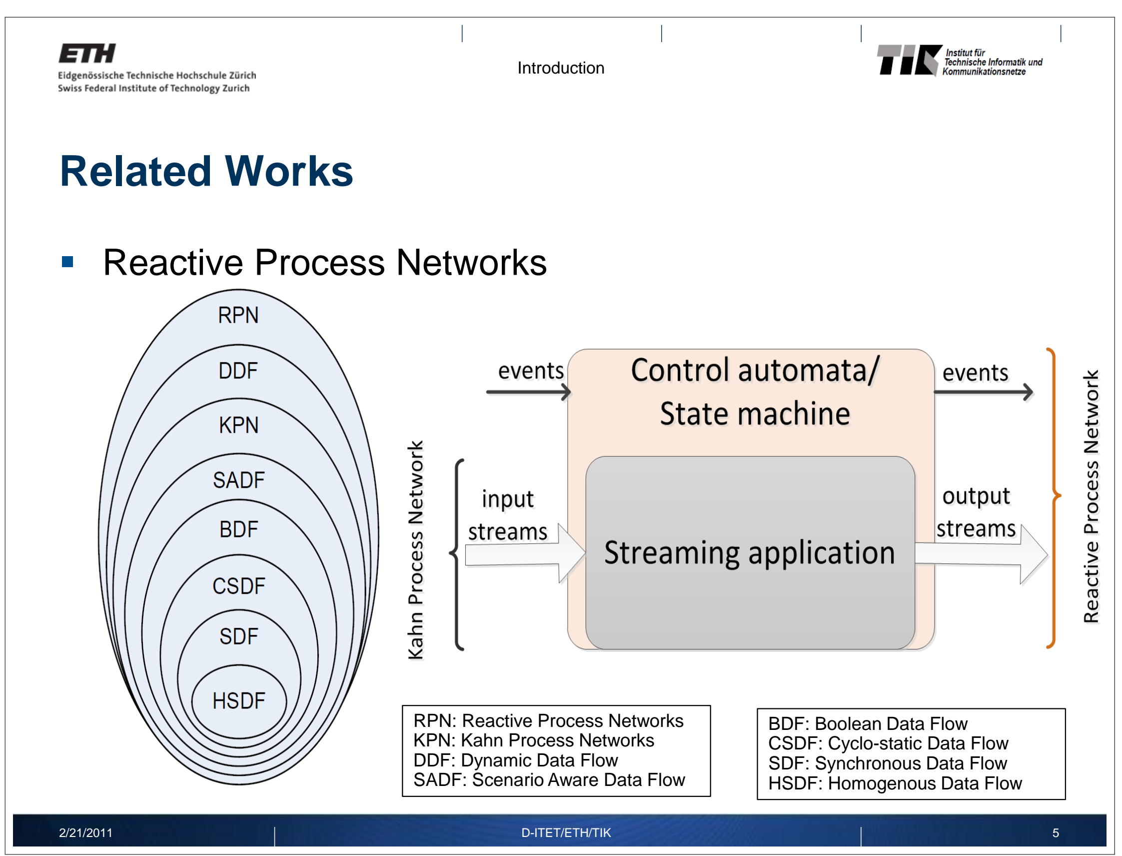
Contents

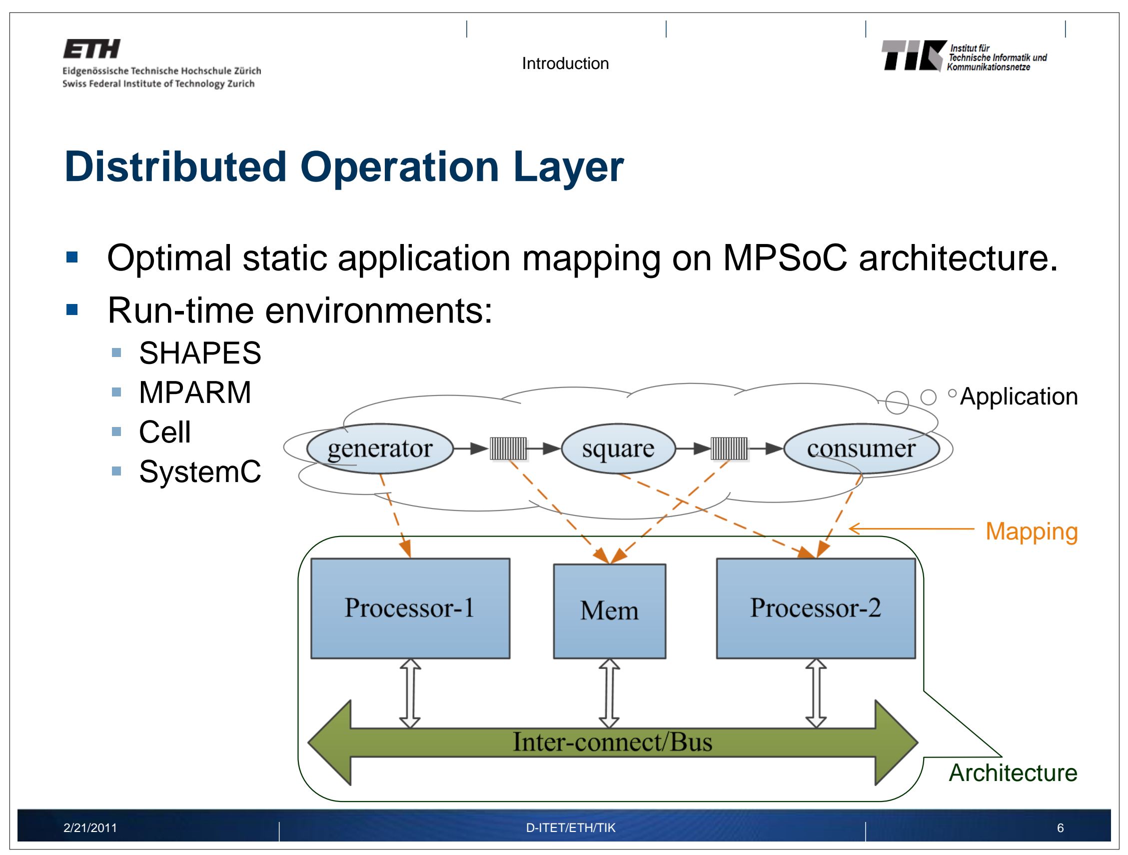
- Introduction
  - Multiple applications
  - Related works
  - Distributed Operation Layer
  - Distributed SystemC simulation
- Dynamism and Distributed Application Layer
  - Dynamism
    - Multiple applications
    - Task remapping
  - Fault tolerance
- Experiments
- Conclusion

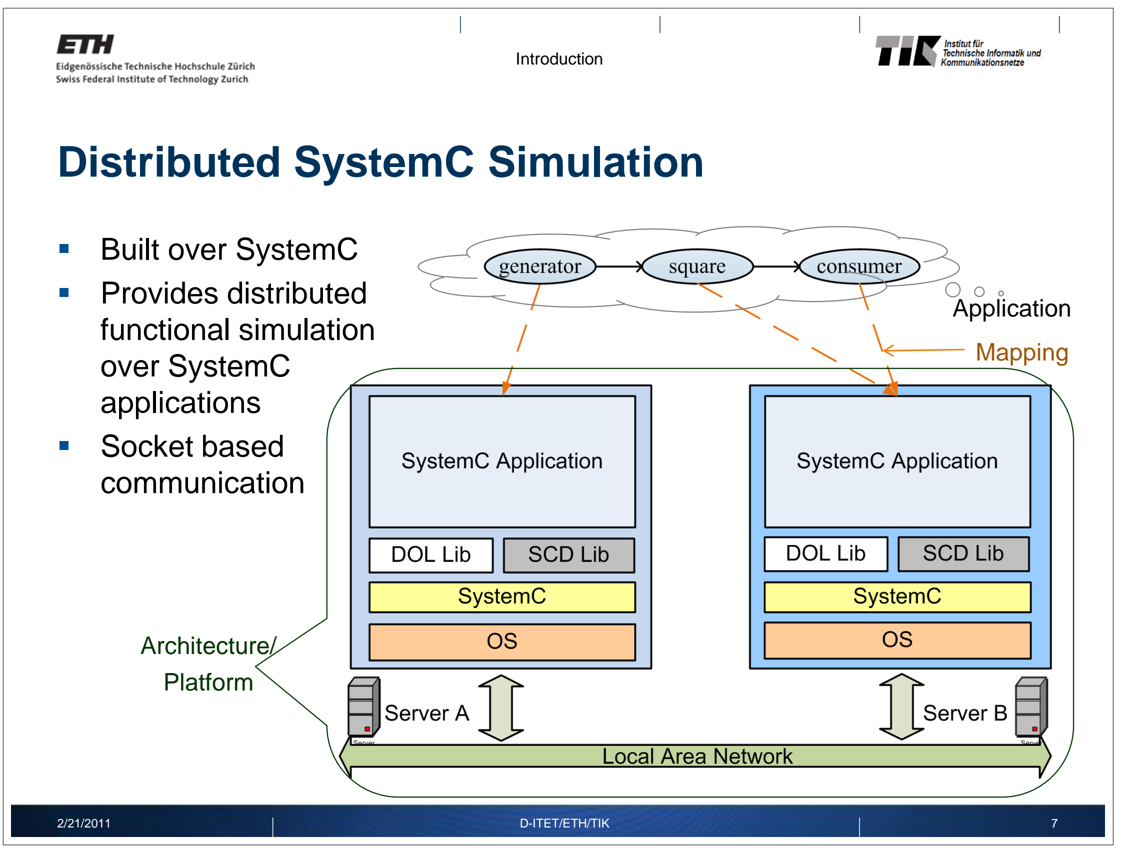
2/21/2011 D-ITET/ETH/TIK 2

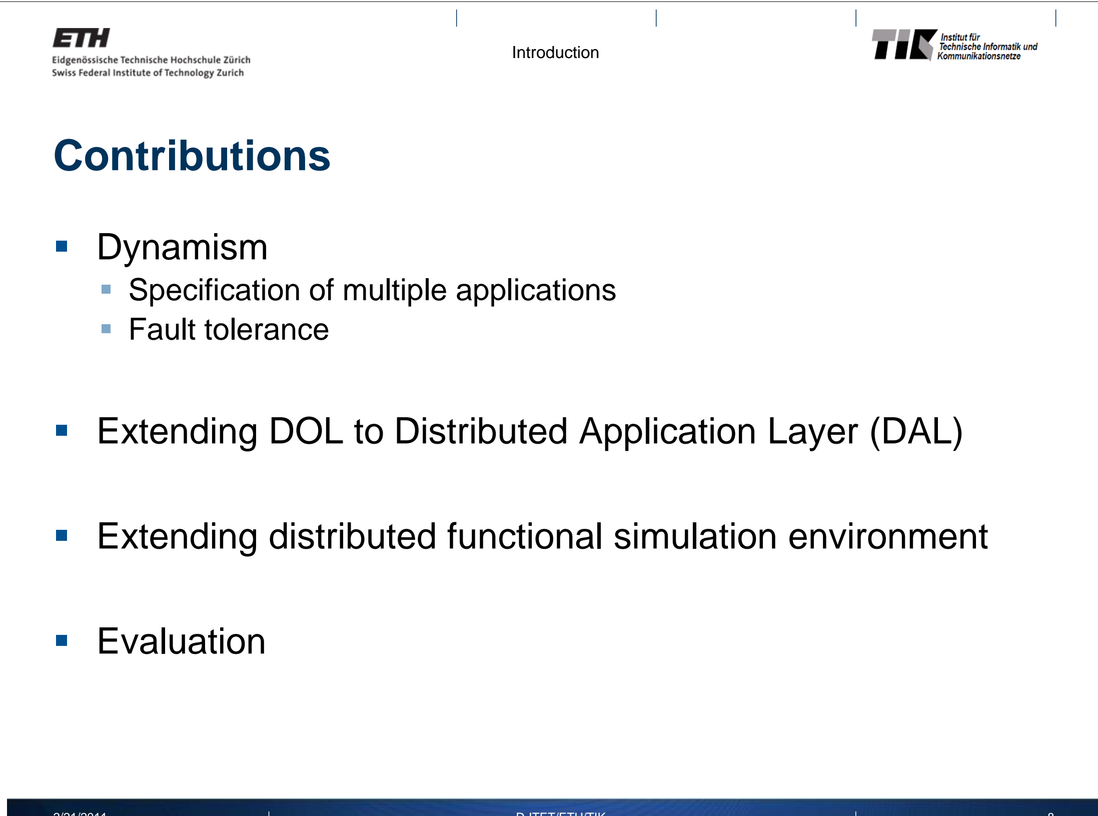


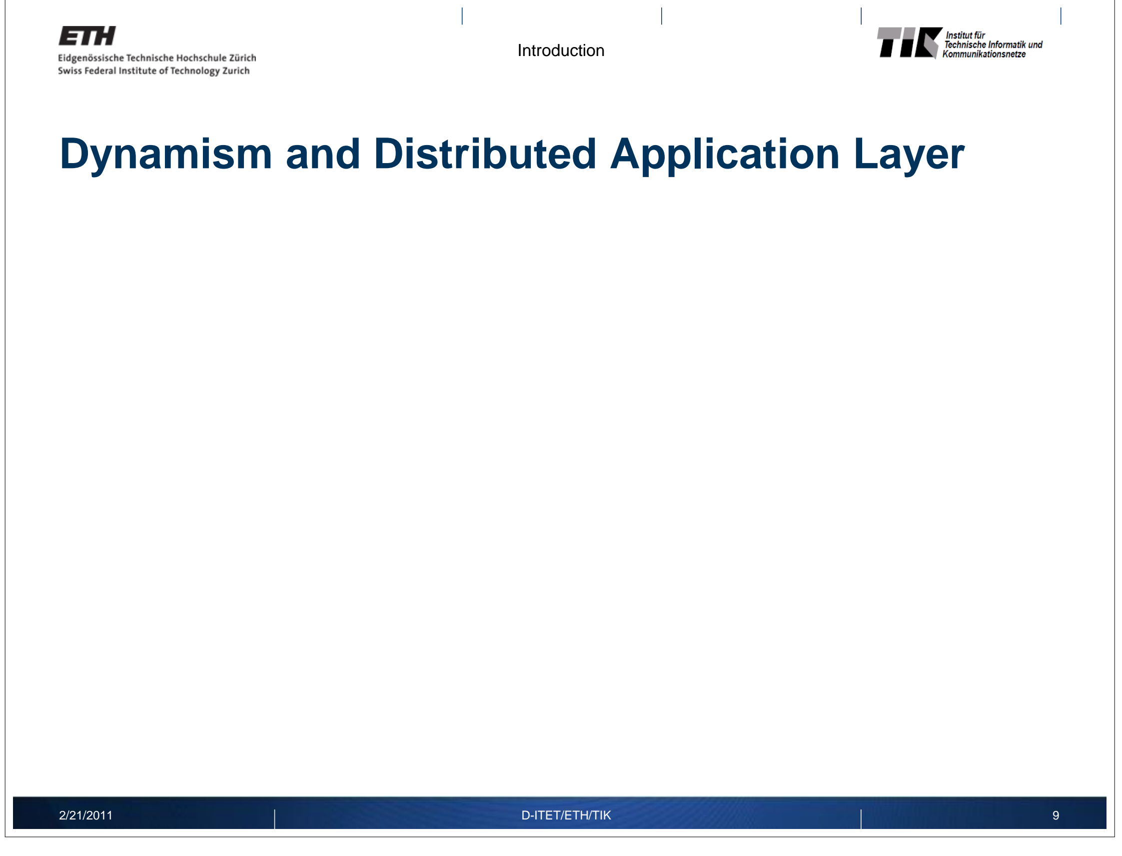
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
Institut für Technische Informatik und Kommunikationsnetze
Multiple Applications: Why?
Multi-tasking
4G
NFC
Adobe
2/21/2011
D-ITET/ETH/TIK
4

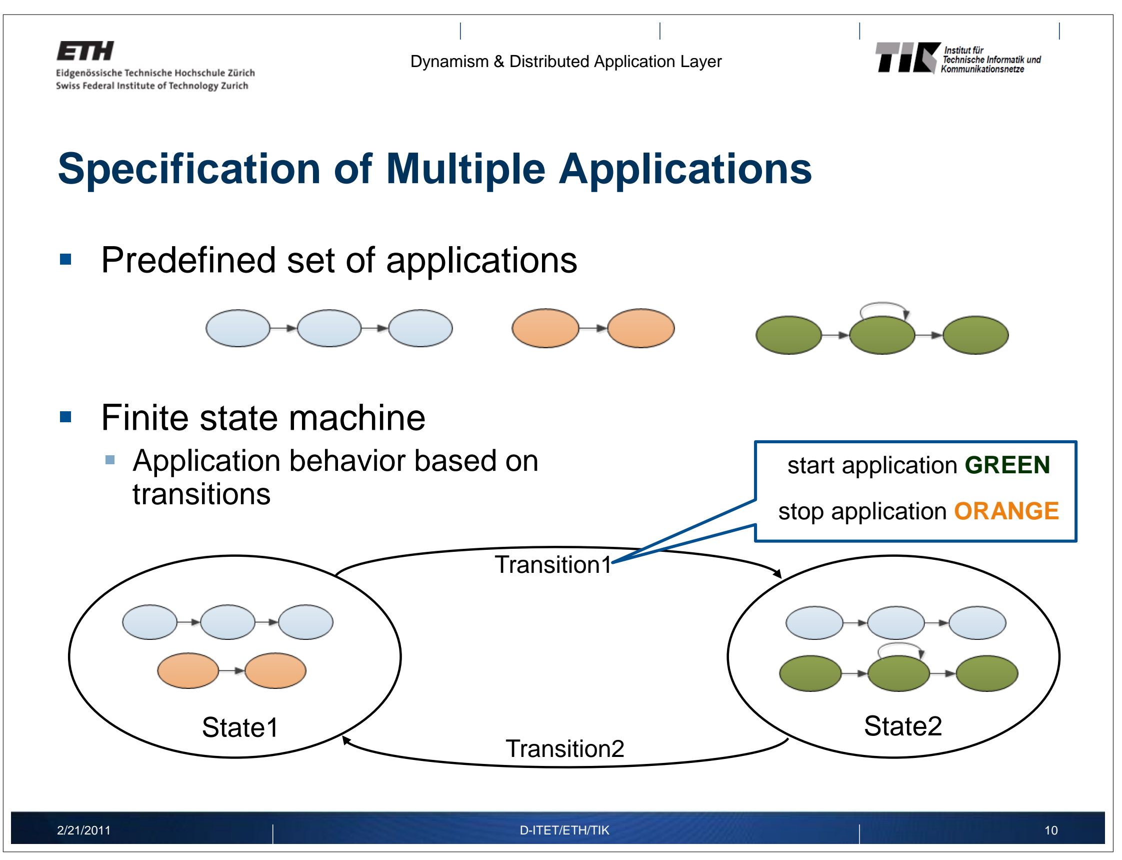

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
Institut für Technische Informatik und Kommunikationsnetze
Related Works
Reactive Process Networks
RPN
DDF
KPN
SADF
BDF
CSDF
SDF
HSDF
events
Control automata/
State machine
events
Kahn Process Network
input
streams
Streaming application
output
streams
Reactive Process Network
RPN: Reactive Process Networks
KPN: Kahn Process Networks
DDF: Dynamic Data Flow
SADF: Scenario Aware Data Flow
BDF: Boolean Data Flow
CSDF: Cyclo-static Data Flow
SDF: Synchronous Data Flow
HSDF: Homogenous Data Flow
2/21/2011
D-ITET/ETH/TIK
5

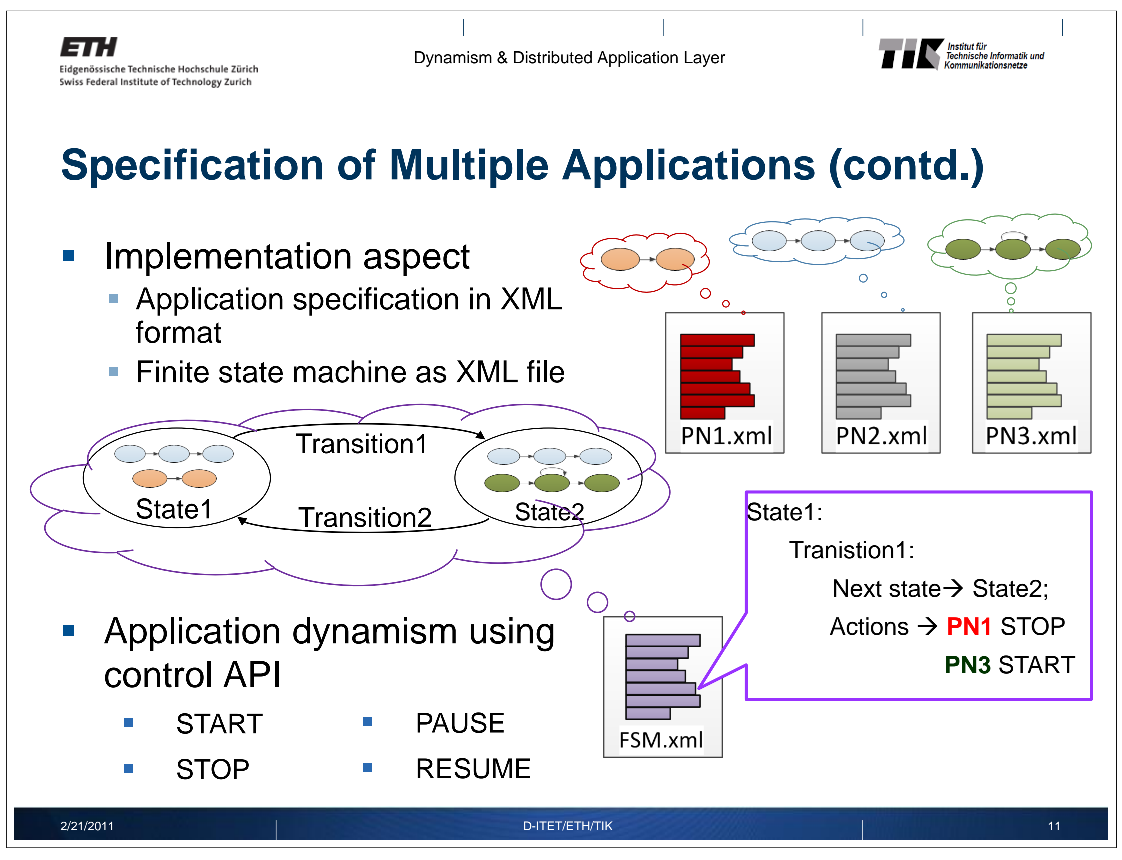

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
TIK
Institut für Technische Informatik und Kommunikationsnetze
Distributed Operation Layer
Optimal static application mapping on MPSoC architecture.
Run-time environments:
SHAPES
MPARM
Cell
SystemC
Application
generator
square
consumer
Mapping
Processor-1
Mem
Processor-2
Inter-connect/Bus
Architecture
2/21/2011
D-ITET/ETH/TIK
6

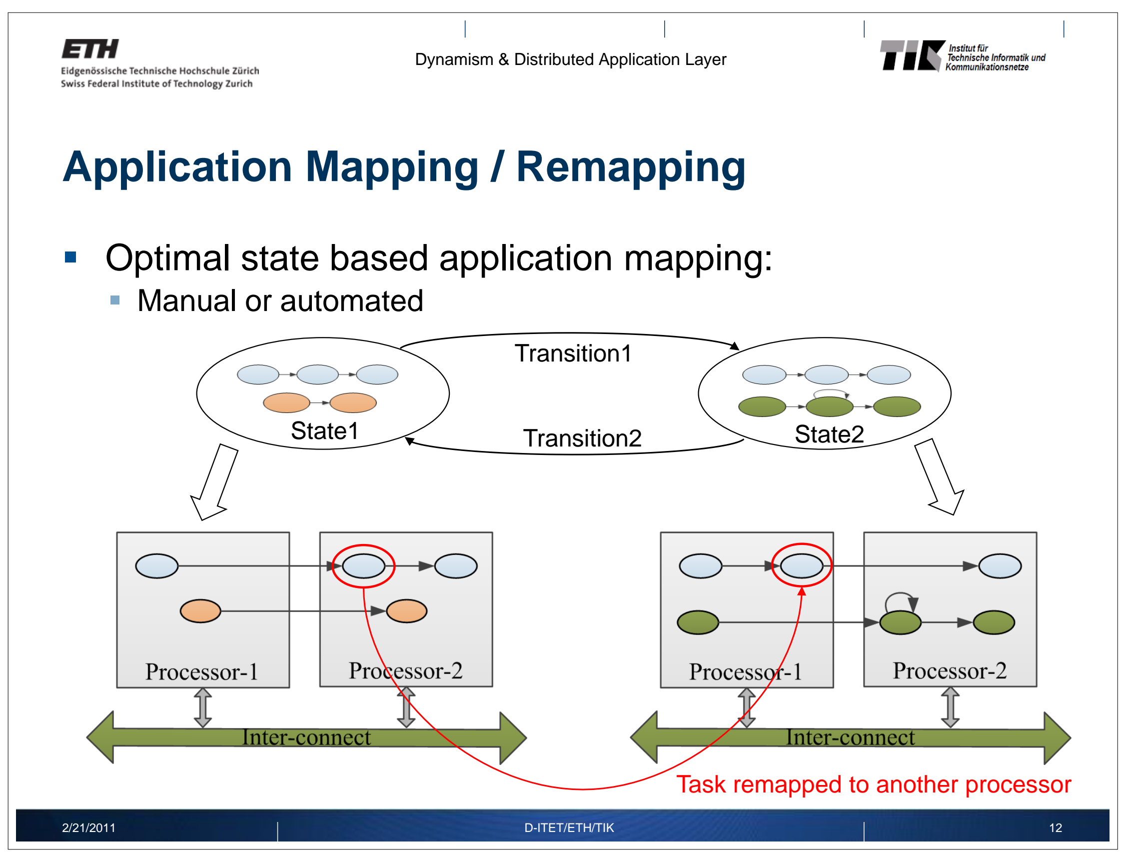

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
TIK
Institut für Technische Informatik und Kommunikationsnetze
Distributed SystemC Simulation
Built over SystemC
Provides distributed functional simulation over SystemC applications
Socket based communication
generator
square
consumer
Application
Mapping
SystemC Application
DOL Lib
SCD Lib
SystemC
OS
SystemC Application
DOL Lib
SCD Lib
SystemC
OS
Architecture/
Platform
Server A
Server B
Local Area Network
2/21/2011
D-ITET/ETH/TIK
7

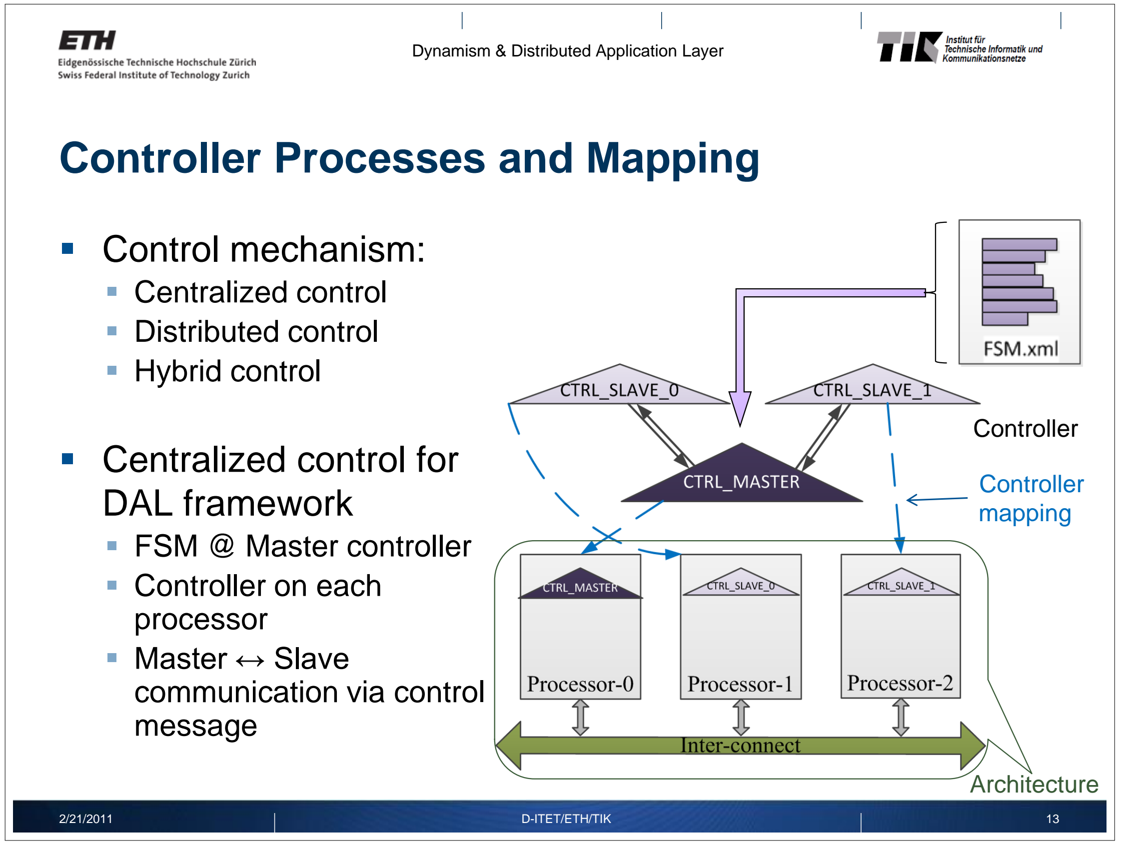

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
TIK Institut für Technische Informatik und Kommunikationsnetze
Contributions
Dynamism
Specification of multiple applications
Fault tolerance
Extending DOL to Distributed Application Layer (DAL)
Extending distributed functional simulation environment
Evaluation
2/21/2011
D-ITET/ETH/TIK
8

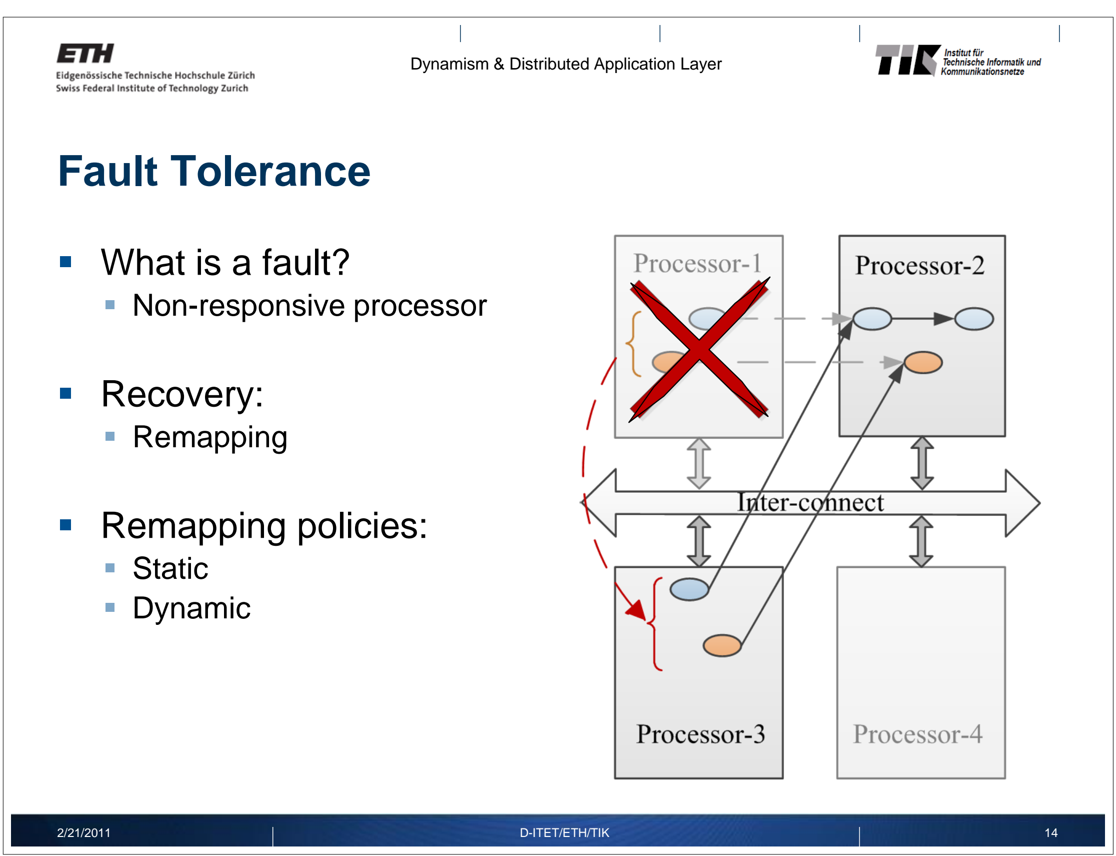

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Introduction
TIK Institut für Technische Informatik und Kommunikationsnetze
Dynamism and Distributed Application Layer
2/21/2011
D-ITET/ETH/TIK
9

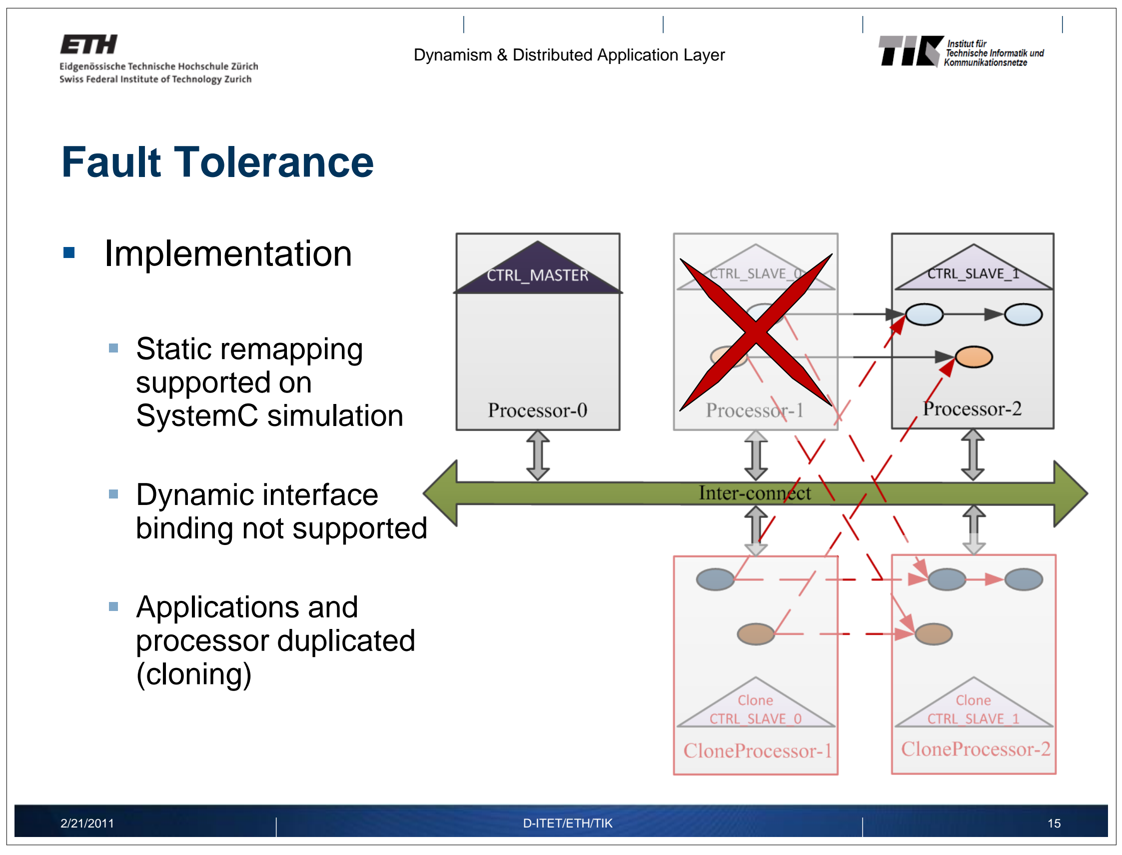

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
TIK
Institut für Technische Informatik und Kommunikationsnetze
Specification of Multiple Applications
Predefined set of applications
Finite state machine
Application behavior based on transitions
start application GREEN
stop application ORANGE
Transition1
State1
State2
Transition2
2/21/2011
D-ITET/ETH/TIK
10


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
TIK
Institut für Technische Informatik und Kommunikationsnetze
Specification of Multiple Applications (contd.)
Implementation aspect
Application specification in XML format
Finite state machine as XML file
PN1.xml
PN2.xml
PN3.xml
Transition1
State1
Transition2
State2
State1:
Tranistion1:
Next state→ State2;
Actions → PN1 STOP
PN3 START
Application dynamism using control API
FSM.xml
START
PAUSE
STOP
RESUME
2/21/2011
D-ITET/ETH/TIK
11


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
Institut für Technische Informatik und Kommunikationsnetze
Application Mapping / Remapping
Optimal state based application mapping:
Manual or automated
Transition1
State1
Transition2
State2
Processor-1
Processor-2
Inter-connect
Processor-1
Processor-2
Inter-connect
Task remapped to another processor
2/21/2011
D-ITET/ETH/TIK
12


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
Institut für Technische Informatik und Kommunikationsnetze
Controller Processes and Mapping
Control mechanism:
Centralized control
Distributed control
Hybrid control
Centralized control for DAL framework
FSM @ Master controller
Controller on each processor
Master ↔ Slave communication via control message
FSM.xml
CTRL_SLAVE_0
CTRL_SLAVE_1
Controller
CTRL_MASTER
Controller mapping
CTRL_MASTER
CTRL_SLAVE_0
CTRL_SLAVE_1
Processor-0
Processor-1
Processor-2
Inter-connect
Architecture
2/21/2011
D-ITET/ETH/TIK
13


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
Institut für Technische Informatik und Kommunikationsnetze
Fault Tolerance
What is a fault?
Non-responsive processor
Recovery:
Remapping
Remapping policies:
Static
Dynamic
Processor-1
Processor-2
Inter-connect
Processor-3
Processor-4
2/21/2011
D-ITET/ETH/TIK
14

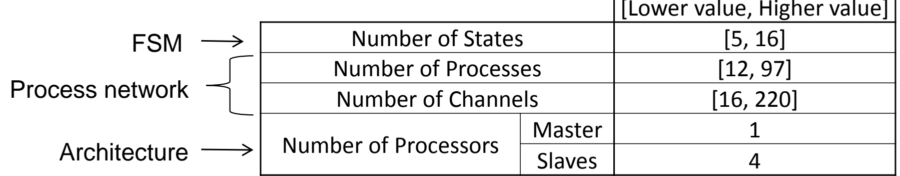

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dynamism & Distributed Application Layer
Institut für Technische Informatik und Kommunikationsnetze
Fault Tolerance
Implementation
Static remapping supported on SystemC simulation
Dynamic interface binding not supported
Applications and processor duplicated (cloning)
CTRL_MASTER
Processor-0
Processor-1
CTRL_SLAVE_1
Processor-2
Inter-connect
Clone
CTRL_SLAVE_0
CloneProcessor-1
Clone
CTRL_SLAVE_1
CloneProcessor-2
2/21/2011
D-ITET/ETH/TIK
15

Experiments

- Application scenarios:

Example	Applications
1	PSC, PMC
2	PSC, PMC, NoC
3	PSC, PMC, FFT
4	PSC, PMC, FFT, Filter
5	PSC, PMC, FFT, Filter, MPEG-2

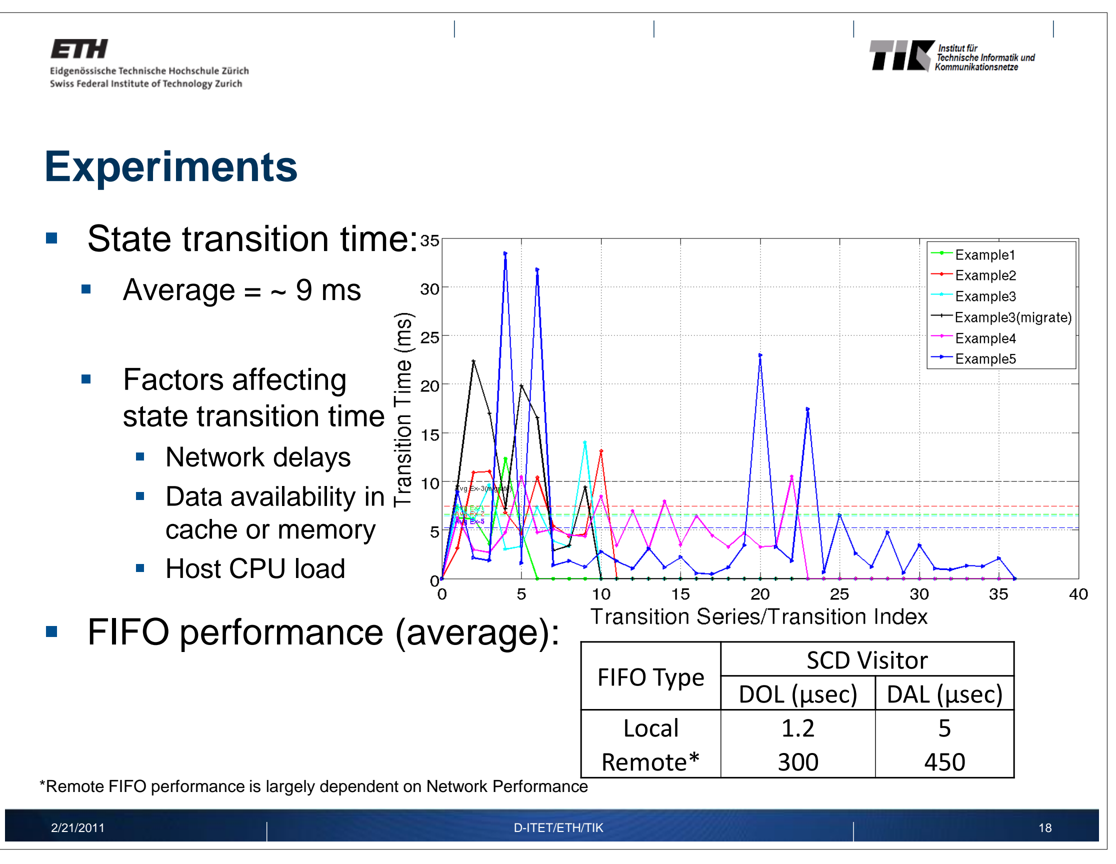
- Statistics:

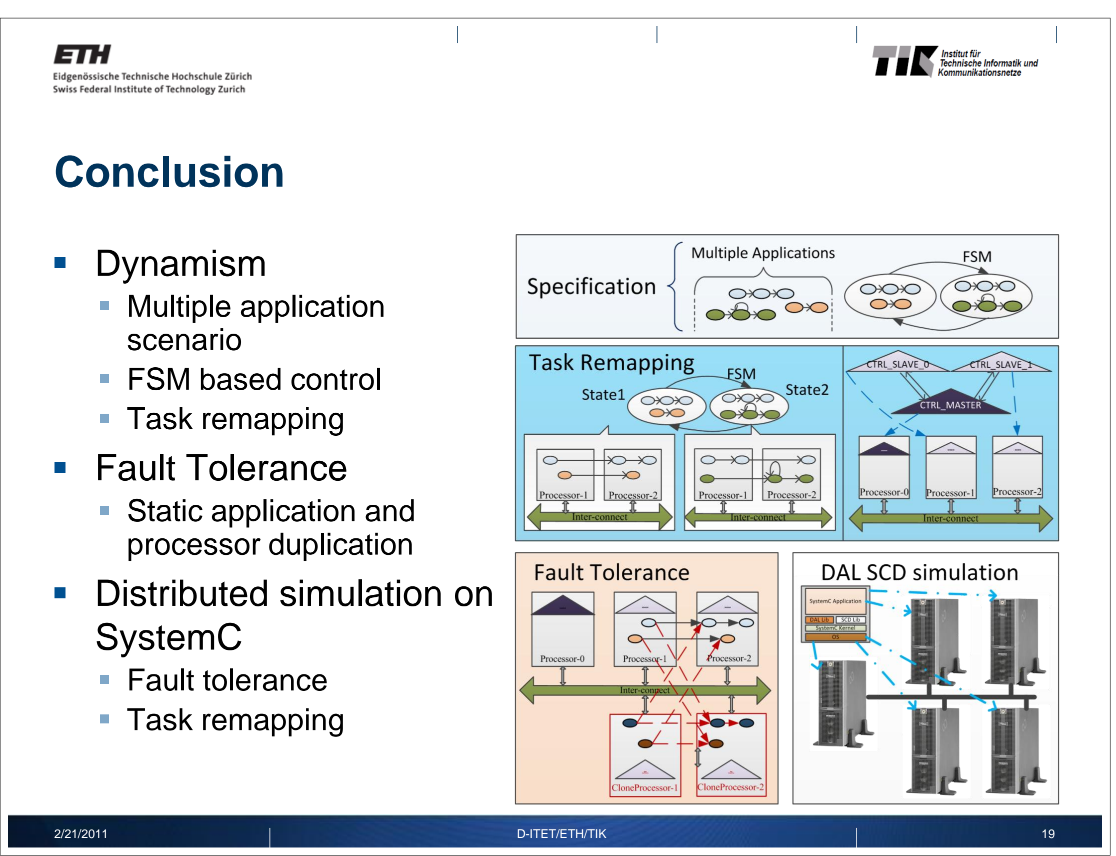


				[Lower value, Higher value]
FSM →	Number of States			[5, 16]
Process network	Number of Processes			[12, 97]
	Number of Channels			[16, 220]
Architecture →	Number of Processors		Master	1
			Slaves	4

- Scalable

2/21/2011 D-ITET/ETH/TIK 17


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze
Experiments
State transition time:
Average = ~ 9 ms
Factors affecting state transition time
Network delays
Data availability in cache or memory
Host CPU load
Transition Time (ms)
Transition Series/Transition Index
Example1
Example2
Example3
Example3(migrate)
Example4
Example5
FIFO performance (average):
FIFO Type
SCD Visitor
DOL (µsec)
DAL (µsec)
Local
1.2
5
Remote*
300
450
*Remote FIFO performance is largely dependent on Network Performance
2/21/2011
D-ITET/ETH/TIK
18


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze
Conclusion
Dynamism
Multiple application scenario
FSM based control
Task remapping
Fault Tolerance
Static application and processor duplication
Distributed simulation on SystemC
Fault tolerance
Task remapping
Specification
Multiple Applications
FSM
Task Remapping
State1
State2
CTRL_SLAVE_0
CTRL_SLAVE_1
CTRL_MASTER
Processor-0
Processor-1
Processor-2
Inter-connect
Fault Tolerance
CloneProcessor-1
CloneProcessor-2
DAL SCD simulation
2/21/2011
D-ITET/ETH/TIK
19

List of Acronyms

ACET	Average Case Execution Time
ARM	Advance RISC Machine
AMBA	Advanced Micro-controller Bus Architecture
BCET	Best Case Execution Time
CBE	Cell Broadband Engine
DAL	Distributed Application Layer
DMA	Direct Memory Access
DNP	Distributed Network Processor
DOL	Distributed Operation Layer
DSP	Digital Signal Processor
EURETILE	European Reference Tiled-architecture Experiment
FIFO	First-In First-Out
FSM	Finite State Machine
HdS	Hardware dependent Software
HdSD	Hardware dependent Software Distributed
KPN	Kahn Process Network
MTTR	Mean Time To Recover
MISO	Multiple In Single Out
MJPEG	Motion JPEG
MPARM	Mulit-Processor ARM
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OS	Operating System
OSCI	Open SystemC Initiative
RISC	Reduced Instruction Set Computer
RTEMS	Real-Time Executive for Multiprocessor Systems
SCD	SystemC Distributed
SDF	Synchronous Data Flow
SHAPES	Scalable Software/Hardware Architecture Platform for Embedded Systems
SIMO	Single In Multiple Out
SIMD	Single-Instruction, Multiple-Data
VLIW	Very Large Instruction Word
WCET	Worst Case Execution Time

Bibliography

[1] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Proc. 7th Int'l Conference on Application of Concurrency to System Design (ACSD)*, Bratislava, Slovak Republic, Jul. 2007, pp. 29–40.

[2] W. Haid, K. Huang, I. Bacivarov, and L. Thiele, "Multiprocessor SoC Software Design Flows," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 64–71, 2009.

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[4] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The Design and Implementation of a First-generation CELL Processor," in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, Feb. 2005.

[5] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing*, 1974.

[6] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A Retargetable Parallel-Programming Framework for MPSoC," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, pp. 1–18, Jul. 2008.

[7] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A Framework for Rapid System-level Exploration, Synthesis and Programming of Multimedia MPSoCs," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '07. New York, NY, USA: ACM, 2007, pp. 9–14.

[8] A. D. Pimentel, C. Erbas, and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Transactions on Computers*, vol. 55, pp. 99–112, 2006.

[9] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based Design Methodology for Digital Signal Processing Systems," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 15–15, Jan. 2007.

[10] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An integrated framework for MPSoC application parallelization," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, Jun. 2008, pp. 754–759.

[11] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, "Trace-based KPN Composability Analysis for Mapping Simultaneous Applications to MPSoC Platforms," in *Design, Automation Test in Europe Conference Exhibition, DATE-2010*, Mar. 2010, pp. 753–758.

[12] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, pp. 1–24, Jan. 2009.

[13] O. Moreira, F. Valente, and M. Bekooij, "Scheduling Multiple Independent Hard-real-time jobs on a Heterogeneous Multiprocessor," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, ser. EMSOFT '07. New York, NY, USA: ACM, 2007, pp. 57–66.

[14] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, Jun. 2007, pp. 777–782.

[15] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha, "Analyzing Composability of Applications on MPSoC Platforms," *Journal of Systems Architecture*, vol. 54, no. 3-4, pp. 369–383, 2008, system and Network on Chip.

[16] A. J. Martin, "The Probe: An Addition to Communication Primitives," Califronia Institute of Technology, Pasadena, CA, USA, Tech. Rep., Jan. 1985.

[17] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J. Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, "YAPI: Application Modeling for Signal Processing Systems," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: ACM, 2000, pp. 402–405.

[18] E. A. Lee and et al., "Overview of the Ptolemy Project," Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, USA, Tech. Rep., Mar. 2001.

[19] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState-An Internal Design Representation for Codesign," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524–544, Aug. 2001.

[20] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, no. 6, pp. 742–760, Jun. 1999.

[21] S. Neuendorffer and E. Lee, "Hierarchical Reconfiguration of Dataflow Models," in *Proceedings of Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004 (MEMOCODE 04)*, Jun. 2004, pp. 179–188.

[22] C. Zebelein, J. Falk, C. Haubelt, J. Teich, and R. Dorsch, "Efficient High-Level Modeling in the Networking Domain," in *Design, Automation Test in Europe Conference Exhibition, DATE 2010*, Mar. 2010, pp. 1189–1194.

[23] M. Geilen and T. Basten, "Reactive Process Networks," in *Proceedings of the 4th ACM international conference on Embedded software*, ser. EMSOFT '04. New York, NY, USA: ACM, 2004, pp. 137–146.

[24] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study," in *Proceedings of the conference on Design, Automation and Test in Europe: Proceedings*, ser. DATE 2006. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 15–20.

[25] A. Aguiar, S. J. Filho, T. G. dos Santos, C. sar Marcon, and F. Hessel, "Architectural Support for Task Migration Concerning MPSoC," in *Proc. Anais do XXVIII Congresso da SBC, WSO Workshop de Sistemas Operacionais (SBC)*, Belém do Pará, PA, Jul. 2008.

[26] H. Shen and F. Pétrot, "Novel task migration framework on configurable heterogeneous MPSoC platforms," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 733–738.

[27] A. Avižieni, "Fault-Tolerance: The Survival Attribute of Digital Systems," in *Proceedings of the IEEE*, vol. 1, no. 10, Oct. 1978, pp. 1109–1127.

[28] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, “Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip,” in *Handbook of Signal Processing Systems*, 2010, pp. 1007–1040.

[29] “EURETILE,” Jan. 2011. [Online]. Available: http://www.euretile.eu/

[30] B. Selic, “The Pragmatics of Model-driven Development,” *Software, IEEE*, vol. 20, no. 5, pp. 19–25, 2003.

[31] “SHAPES,” May 2010. [Online]. Available: http://www.shapes-p.org/

[32] T. Sporer, M. Beckinger, A. Franck, I. Bacivarov, W. Haid, K. Huang, L. Thiele, P. S. Paolucci, P. Bazzana, P. Vicini, J. Ceng, S. Kraemer, and R. Leupers, “SHAPES: A Scalable Parallel HW/SW Architecture Applied to Wave Field Synthesis,” in *Proc. 32nd Int'l Audio Engineering Society (AES) Conference*, Hillerød, Denmark, Sep. 2007, pp. 175–187.

[33] L. Benini, “MPARM,” Jun. 2004. [Online]. Available: http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

[34] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, “MPARM: Exploring the Multi-Processor SoC Design Space with SystemC,” *The Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, Sep. 2005.

[35] “RTEMS Operating System,” Jul. 2010. [Online]. Available: http://www.rtems.com/

[36] *RTEMS ARM Applications Supplement*, 4th ed., On-Line Applications Research Corporation, Aug. 2003.

[37] K. Huang, W. Haid, I. Bacivarov, and L. Thiele, “Coupling MPARM with DOL,” ETH Zürich, TIK Report 314, Sep. 2009.

[38] “Protothreads: Lightweight, Stackless threads in C,” Oct. 2006. [Online]. Available: http://www.sics.se/~adam/pt/

[39] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, “Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs,” in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Grenoble, France: IEEE, 2009, pp. 35–44.

[40] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, “A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach,” in *Embedded Processor Design Challenges*, ser. Lecture Notes in Computer Science, E. Deprettere, J. Teich, and S. Vassiliadis, Eds. Springer Berlin / Heidelberg, 2002, vol. 2268, pp. 321–324.

[41] “SystemC Community Website,” Aug. 2010. [Online]. Available: http://www.systemc.org/community/

[42] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Boston, USA: Kluwer Academic Publishers, 2004.

[43] F. Hugelshofer, “Scalable Distributing Mutli-Linux System for DOL Application,” Master's thesis, ETH Zurich, Zurich, Switzerland, Jan. 2008.

[44] “IEEE Standard SystemC®Language Reference Manual,” IEEE Computer Society, Design Automation Standard Committee, 2006.

[45] “OSCI: Open SystemC Initiative,” Jan. 2011. [Online]. Available: http://www.systemc.org/

[46] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, “Scalably Distributed SystemC Simulation for Embedded Applications,” International Symposium on Industrial Embedded Systems, 2008, pp. 271–274.

[47] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[48] R. W. Stevens and S. A. Rago, *Advanced Programming in the UNIX® Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[49] “SAX:About SAX,” Jan. 2011. [Online]. Available: http://sax.sourceforge.net/

[50] “JDOM,” Jan. 2011. [Online]. Available: http://www.jdom.org/

[51] J. Hunter, “JDOM and XML Parsing, Part-1,” *Oracle Magazine*, pp. 68–73, Oct. 2002.

[52] ——, “JDOM and XML Parsing, Part-2,” *Oracle Magazine*, pp. 70–72, Nov. 2002.

[53] W. Haid, K. Huang, and S. Künzli, “Application Examples,” Jul. 2008.

[54] S. Mall, “MPEG-2 Decoder for SHAPES DOL,” Master’s thesis, ETH Zurich, Zurich, Switzerland, Apr. 2007.

[55] W. Haid and K. Huang, “Schematics of the DOL Schemata,” Jul. 2008.

[56] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced Features of the Message Passing Interface*, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[57] M. Saldana and P. Chow, “TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs,” in *Field Programmable Logic and Applications, 2006. FPL ’06. International Conference on*, 2006, pp. 1–6.

[58] I. Labs, “The SCC Programmers Guide,” May 2010. [Online]. Available: techresearch.intel.com/spaw2/uploads/files/SCCProgrammersGuide.pdf