The Department of Electrical and Computer Engineering
The University of Auckland
New Zealand

Designing GALS software systems using libraries and run-time OS support

Wei-Tsun Sun
August 2012

Supervisors: Prof. Zoran Salcic
Dr. Morteza Biglari-Abhari

A Thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Abstract

With the increasing use of multicore and distributed computing platforms, software systems are becoming more and more complex and as such require tremendous design effort. They are also very difficult to debug and guarantee for correct functionality. In the case of embedded systems, ideally, software development could proceed in parallel with the development of the target hardware if this is not known in advance.

This thesis addresses complex software systems development which can be underpinned by formal models of computation and which use some kind of operating system to abstract the hardware platform from system developers. Two specific models of computation, synchronous reactive and asynchronous, combined into a Globally Asynchronous Locally Synchronous (GALS) model are used as the underlying formal model of the target systems. A set of tools to implement the GALS model in traditional programming languages, C and C++, is used to enable re-use of huge legacy codes. The tools consist of libraries and run-time support that allow the design of two types of GALS systems for the range of target platforms: (1) static systems with a fixed number of concurrent processes and (2) dynamic GALS (DGALS) systems where the number of processes varies during system life. The implemented libraries and run-time support depend only minimally on the operating system, since they use a very primitive synchronization mechanism in the form of semaphores, and are ported to a number of non-real-time and real-time operating systems with identical application programming interface (API).

A specific version of API is developed for the development of static GALS systems in system-level design language SYSTEMC, which allows system designers to model both hardware and software within the same system model, thus developing software before the actual hardware is available.

The developed APIs are in compliance with the GALS model of computation (MoC), opening the possibilities for formal verification of designs or their parts, or of the use of the API in conjunction with programming languages based on GALS MoC.

Acknowledgements

I have to express my gratitude to my supervisor, Professor Zoran Salcic, for his guidance, inspiration, and support. Valuable suggestions and directions are always provided when I needed them the most. Having discussions throughout the research strengthened my analytical skills and ability to see things at different angles. He always asked me to see a full picture which helped me to obtain the right direction of not only in research but also aspects in life.

Knowing and being supervised by Professor Salcic has changed my life. It began from that he opened the door of pursuing further knowledge for me, when I was supervised by him for my final year project, in my last year of undergraduate study. I admire Professor Salcic, for both of his wisdom and knowledge, as well as the enthusiasm. His kind support and caring continue to help me overcome many challenges and difficulties in my life.

I would like to thank my family from the bottom of my heart for their unconditional support. My father, Kai-Ping Sun, a civil engineer, has always inspired me. I would like to thank my mother, Pi-Chu Tu, for her care and love. Brother, Pei-Tsun Sun, I thank you for your company and humor that cheered me often.

Furthermore, I would like to thank Dr. Avinash Malik and Dr. Alain Girault for their suggestions and helps toward the research. Many thanks to Dr. Morteza Biglari-Abhari and Dr. Partha Roop for the support they have offered. I would also like to thank many other staff from the department of Electrical and Computer Engineering for both work and technical support.

I must thank my friends and colleagues whom I have worked with throughout these years. Your presence is vital, not only during my research, but also my life. I thank you all.

子曰：“學如不及，猶恐失之。”

The above is a quote from the sayings of Confucius, the Chinese philosopher (511 BC – 479 BC). It means to learn as your goal will never be achieved; worry about not keeping yourself updated. I dedicate this thesis to myself, who came to the point of understanding this saying throughout the research work and thesis writing. This thesis is not the destination of my research. It is the telescope for me to see the universe of knowledge I would continue to pursue.

Table of Contents

List of Figures

List of Tables

List of Listings

1

Introduction

The development of computer systems is driven by technology advancements. New application requirements, particularly those that can be classified as embedded systems, are becoming very challenging due to increased system complexity. The large number of concurrent behaviors that are implemented in combination with hardware and software components require us to change traditional design practices to reflect these new realities. The development of systems is typically divided into phases of system specification, system verification, component partitioning, simulation, implementation, and validation, before the delivering of the final product.

1.1 Problems in design of computer systems

Designing computer systems requires sophisticated techniques to overcome constraints caused by the complexities of the systems under design. It is expected that complex system design requires much time. However, to ensure that products will be on the shelves on time, extra effort must be spent within the time available for product release. There are methodologies and remedies to address the underlying issues, for example design-gap, by adopting the use of advanced system modeling and synthesis techniques. System models often include models of embedded and/or real-time operating systems (OSs), to estimate and mimic dynamic behaviors in the final systems.

Because of the ever-increasing application domains of computer systems and time-to-market constraints, and in order to maintain maximum productivity, current design methodologies focus on how to achieve the final implementation in the shortest time with the minimum resources. There are many kinds of computer systems, performing different dedicated operations. For example, automation (in manufacturing), transportation (automotive applications and traffic control), communications (e.g. mobile phones and internets), and healthcare (electronic aids and life support) are typical applications of computer systems. The design approaches and tools available are often application-field specific. Such approaches sometimes come from experience; i.e. they are heuristic. Designs following such methodologies may work without any problem for long periods of time, but because of a lack of theoretical background and support for analysis, hidden problematic issues are hard to detect and locate when error occurs. Safety and liveness are two important aspects of critical systems; such systems need to be designed with care and should be tolerant to unforeseen events (fault tolerant). Without proper theoretical-based reasoning, designing critical systems may be just like filling visible holes which is an unreliable approach. For instance, thread-based designs in software programming are error prone [Oracle, 1999], [Lee, 2006], and difficult to program [Serrano et al., 2004] because they rely on the experience and care of the designer.

Furthermore, as technologies advance, the performance of processing units (i.e. processors) and available connectivity (e.g. high speed networks) are widely available. The scope of a system under design is no longer a single chip or computer but also networks of distributed computational nodes. The size of each computational node varies according to its requirements and the operations that it performs. However, the design still depends on the background of the designer, so that one may not have the full picture of the system under design, and the coverage of the thinking is just not wide enough. The consequences may not be just the failure of the system, but also of unpredicted design time and financial losses.

1.2 Tools based on theory to provide correct designs

Tools and methodologies based on theories have been proposed and developed widely to provide designers peace of mind when carrying out design tasks and decisions. The approaches are mostly application-specific, similar to heuristic methods but better in the sense that potential problems can be found with given limitations and requirements. Limitations in methodological approaches are not necessarily drawbacks for the designs, but can sometimes be ground rules to prevent incorrect plans and strategies which may lead to a disaster in the design process.

The focus of this thesis is to address the issues and to provide the tools with theoretical basis to design concurrent systems with less design effort to ensure both the safety and the correctness of the design. It is also to enable the linkage of such tools to other application domains with concurrency. Such tools are also suitable but not limited to system-level design, containing components that are executed concurrently.

Existing theoretical methodologies for designing systems with concurrency are based on various models of computations (MoCs). MoCs such as synchronous languages (ESTEREL [Berry & Gonthier, 1988], LUSTRE [Caspi et al., 1987], and SIGNAL [Benveniste et al., 1985]), process calculi (e.g. Communicating Sequential Processes/CSP [Hoare, 1978]) and networks (e.g. KPN [Kahn, 1974]), globally asynchronous and locally synchronous (GALS [Chapiro, 1984]) systems, and process mobility (such as pi-calculus [Milner, 1999] for dynamic systems) have been proposed and developed. Some of these approaches, such as process calculi and networks, are purely theoretical and lack support in design systems. On the other hand, system level design languages (SLDLs), such as SYSTEMC [OSC Initiative, 1999], have a limited level of support for formal MoC, i.e. they are based on the discrete event MoC which does not guarantee determinism. However, they are widely used in the design community and also serve as an industrial standard.

Languages such as SystemJ [Malik et al. 2010] and DSystemJ [Malik et al., 2010] consider both the theoretical and the practical features of designed systems and cover a vast range of emerging systems. However, the requirement of using the Java virtual machine as their target narrows down the door to using them in systems with limited resources. Furthermore, existing SLDLs are closely related to programming languages used in embedded platforms, such as C/C++, which makes the integration between SLDLs and SystemJ/DSystemJ less straightforward.

1.3 Motivation

From current application trends and available design and implementation approaches, it would be desirable to have a set of tools which are able to handle concurrency based on theoretical foundations, but at the same time work closely with existing tools to carry out system design. In order to extend the domain where current tools can be applied effectively it is necessary to address a range of issues. A non-exclusive list of these issues follows:

The need for tools to support formal MoC.

Synchronous languages are not suitable for a distributed platform due to the overheads to maintaining a global sense of instance/tick. Asynchronous languages and libraries are error prone for programming concurrency. The path adopted in this thesis is to use the GALS MoC as the basic formal model that underpins the design approach and tools.

The need for tools to design both control- and data-dominant systems.

Concurrency exists in the realms of both control and data domains. Handling multiple events at the same time correctly and efficiently is required in complex reactive systems. At the same time, multiple data streams are being processed concurrently through the uses of multicore/multiprocessor architectures. Moreover, computer systems (applications) are heterogeneous and consist of a mix of control and data parts.

The need for bridging hardware and software via extensions to operating systems.

Operating systems play the role of bridging between hardware and software components within systems. Device drivers and firmware are parts of the operating systems to control hardware peripherals around the processor. Extensions to the operating systems are necessary to enable hardware/software co-design and co-synthesis where functionalities implemented either in hardware or software are relevant to the operating systems running between them.

Integration to current system level design language (SLDL) to help the design process.

As mentioned before, existing SLDLs lack support for formal MoCs but are popular in both industry and academia. Developing a set of new tools does not imply re-inventing the wheel. The proposed tools can be used to describe software behaviors and should be able to link with current state of the art SLDLs with minimal effort.

Support for distributed and dynamic systems with a small footprint in mind.

The concept of distributed systems is not new. It is desirable to have concurrent programming suitable for distributed systems while a specific MoC is followed. DSystemJ and X10 [Charles et al., 2005] aim for distributed computing, yet both require JVM, which implies higher performance underlying execution platforms. In this thesis, a library-based framework that extends C language is proposed, implemented, and experimentally verified, and considered as a potential solution.

Single-language approach is used to improve productivity.

Designing a system in a single language frees the designer from interfacing components described in different languages which can be prone to mistakes. Synchronous languages such as ESTEREL are difficult for describing software algorithms and require significant efforts for either hardware or software implementation. Because of such limitations, algorithms that perform data computations are implemented in the other host languages such as C/C++ and are linked with the compiled synchronous programs. Approaches such as ECL [Lavagno & Sentovich, 1999] and JESTER

[Antonotti et al., 2000] attempted to achieve a single language environment. However, source codes are still required to be generated to backend ESTEREL language where debugging is carried out.

Simulink [MathWorks, 2011] based on MatLab language, is able to provide code generations and is very flexible for system modeling. However, it is used mainly for system simulation.

Thus the required tools should be based on a single language. However, such a language should still be able to provide interface to bind with other languages to ensure interoperability. The C language is chosen as the lowest-level common denominator and used to implement the required libraries and tools in this thesis.

Create an extension, in the form of a library, to support MoC of existing languages:

Instead of proposing another language, a library-based extension of existing programming language is proposed and introduced. Such an extension is able to fulfill and support the semantics of GALS and DGALS MoC, which have been provided in languages such as SystemJ [Malik et al. 2010], DSystemJ [Malik et al., 2010], and other related languages such as synchronous languages ESTEREL [Berry & Gonthier, 1988]. By following specific MoC, the behaviors of systems modeled in the proposed extension will be deterministic and predictable.

1.4 Research contributions

The research contributions of this thesis are illustrated as two layers of the inner circles in

Figure 1.1The innermost circle in Figure 1.1 represents the key developments of language extensions that enable a design underpinned by a GALS MoC, as well as the methodology to include operating system models in more complex designs. The second layer extends and utilizes the results of the inner layer to create frameworks to improve design productivity in a complex system design. The outer layer presents the fundamental concepts and related approaches that are analyzed and combined into the results of the thesis.

OS modeling is first carried out to pinpoint the required services and programming support for synchronous languages in system level design, and which consider both software and hardware components in the system. Derived and developed from these concepts, libGALS is implemented as a library, based on the basic services provided by the underlying operating system. The DesignGALS framework merges the works of both OS modeling and libGALS to perform system-level design. Similarly, libDGALS, which is a further enhancement and extension of libGALS, supports programming of dynamic GALS systems and provides the mechanisms to model and design distributed systems in a higher level of abstraction.

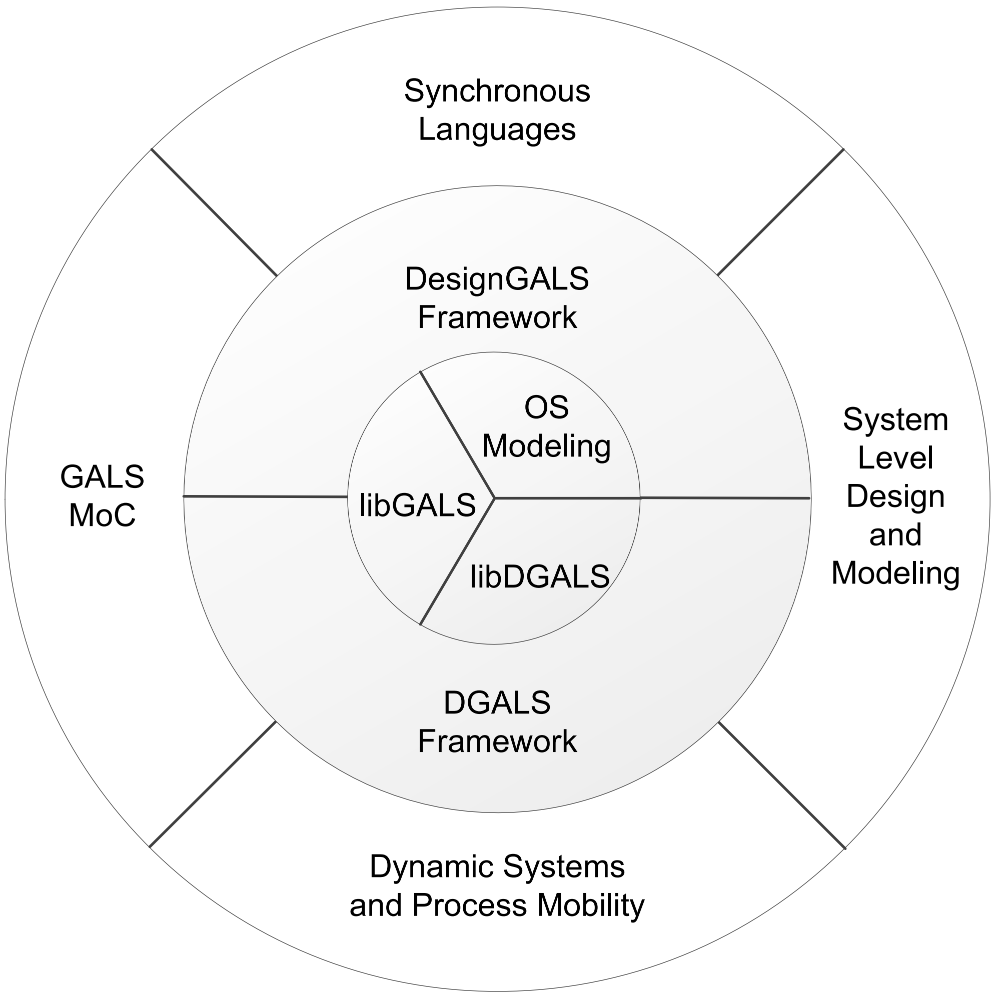


Figure 1.1: Contributions of this thesis in relation to other work

1.5 Thesis organization

The rest of the thesis is organized as follows:

Chapter 2 gives an overview of the related work. It introduces the motivation, contribution and the structure of the thesis. Underlying theoretical and practical concepts relevant to the thesis are given in this chapter. An overview of basic and essential background along with related works is given in Chapter 2, so that the reader can understand the context in which the thesis was written.

Chapter 3 introduces the basic principles of developing computer systems in a staged design flow; collection of design models used in different design phases is presented. SYSTEMC is used as the main system-level design language (SLDL) throughout this chapter and this thesis. Methodologies of modeling software concurrency are also discussed and investigated. A system model that consists of software processes, OS, and hardware components such as data memory and peripherals is proposed. The behaviors of processors in the system under design are presented in a higher level of abstraction through the models of software processes and OS, in contrast to a lower level description such as RTL or simulation performed by ISS. The OS model is the main focus and is used to explore possible run-time supports to describe synchronous concurrency, the basic building blocks of GALS systems.

The findings in Chapter 3 lead to the development of libGALS detailed in Chapter 4, where a library-based approach to describe GALS programs, libGALS, as the run-time support to OSs, is presented. The application programming interface (API) and the internal data structure of libGALS are detailed in this chapter. Examples of constructing a GALS program are given, followed by the experiments and results obtained by comparing libGALS to SystemJ, a language and compiler-based approach for designing GALS systems.

Chapter 5 presents a framework that integrates libGALS and SYSTEMC, called GALS-Designer. GALS-Designer enables the system designer to describe the overall system consisting of GALS software and hardware components in the same SYSTEMC model. How libGALS is integrated with SYSTEMC is detailed in this chapter. System models constructed using GALS-Designer benefit from using a multicore host to obtain simulation speed-up compared to conventional system models in pure SYSTEMC. GALS-Designer allows the instantiation of multiple libGALS programs in a system. Because simulations of GALS programs can be un-timed functional or approximate-timed by having timing annotations, GALS-Designer is suitable in various design phases of the design flow. Case studies of using GALS-Designer are presented and are followed by evaluations of the GALS-Designer.

Based on the introduction to libGALS in Chapter 4, Chapter 6 presents its extension to libDGALS, that clock domains can be created dynamically in distributed networks. This extension to libGALS requires dynamic library-loading and the ability to operate over a network, available in modern operating systems. The libDGALS is implemented according to the Dynamic GALS (DGALS) MoC, and is the backbone of the DynamicGALS framework. The internals of libDGALS will be addressed in Chapter 6, along with case studies and comparisons with other relevant approaches. Corresponding to GALS-Designer in Chapter 5, in which libGALS programs are instantiated statically in the elaboration phase of the SYSTEMC simulation, clock domains in libDGALS programs are created dynamically at run-time.

With Chapter 7, the thesis concludes by summarizing the advantages of the overall framework built around the libGALS library and run-time support to C language. Future directions are also presented.

2

Background and related works

Background information required for a clear view of this thesis is provided in this chapter. A general overview is given to understand the later chapters. Section 2.1 gives brief descriptions of concurrency in computer system design. Typical elements of computer systems are briefly described in Section 2.2. Concurrencies of computer systems are detailed in Section 2.3. This is followed by discussions of models of computations (MoCs), languages, and libraries for system design in Section 2.4. Sections 2.5 to 2.13 give further insights into the current state of the art, leading to the approach taken in this thesis, briefly detailed in Section 2.14.

2.1 Types of computer systems

Computer systems come with different flavors according to different characteristics of their requirements. They can be categorized as 'transformational systems', 'reactive systems' [Harel & Pneuli, 1985], and 'interactive systems' [Raymond et al., 1998] according to how they behave in relation to the external environment.

Transformational systems, also known as data-dominated systems, operate at their own speeds, which can be periodic or aperiodic. These systems usually have data arriving at regular intervals and the information they carry is more critical than their time of arrival. Signal processing is an example of such systems. A compiler can be seen as an example of an aperiodic transformational system. The time that a transformational system takes to complete given tasks depends on the complexity of the computation and how powerful the underlying computer system.

Reactive systems, on the other hand, operate at the speed of the environment, having to respond to events from the environment continuously and fast enough, i.e. before the next event occurs. Being control-dominated, they are suitable for control-based applications such as automotive systems, robots and many other systems. Therefore logical (how to behave) and temporal (when and how fast to behave) correctness of such systems is important.

Interactive systems respond to the environment similarly to reactive systems, but perform at their own speed as transformational systems. Personal computers are examples of interactive systems, where users can be seen as the environment which provides inputs; the outputs (e.g. display and sound) are produced when the computation is finished depending on how fast the systems are.

A computer system can have characteristics from a mixture of these three types of systems. Heterogeneous systems are typically named after the combination of control-dominated and data-dominated systems [Radojevic et al., 2006].

2.1.1 Embedded systems

Embedded systems are computer systems which deal with externally or internally generated events, similar to reactive systems, in synchronous or asynchronous fashion. Events can occur either externally, such as a temperature variation detected by the dedicated sensor, or internally, such as generated time-outs. Embedded systems usually perform in an interchangeable and non-terminating fashion. The major markets of embedded systems include applications in automotive control (steering control, brakes control, radio navigation, doors control, and suspension control, etc.), communications, handheld devices, or aerospace applications [Stepner et al., 1999].

2.1.2 Real-time systems

Computer systems that require attention to complete given tasks within pre-determined timing constraints are real-time systems. These are further categorized into two groups: (1) hard real-time systems and (2) soft real-time systems [Lab, 1999]. Timing constraints must be fulfilled precisely at all times to prevent system failure in hard real-time systems. Concurrent behaviors within hard real-time systems are crafted carefully with approaches to ensure that critical timing requirements are met. Applications of hard real-time systems include safety-critical systems such as air-bags used in automobiles.

In contrast, soft real-time systems may not fail or can recover if failing to respect the timing specification. Applications requiring actions taken in a timely manner, such as the temperature adjustment of air-conditioning systems, are categorized as soft real-time systems.

2.2 Hardware and software in computer systems

Computer systems are becoming very complex and challenging to design, and the process is spread over a number of stages, such as specifying systems, design exploration, implementation, and verification. Because computer systems are often a composition of concurrent behaviors, these behaviors are represented in different forms at each design stage.

A computer system is generally implemented as a combination of hardware (HW) and software (SW). Concurrency of such systems is also implemented in both domains. Hardware circuits are concurrent in nature, while software concurrency is achieved through the uses of compile-time techniques or run-time support from operating systems (OS).

Having concurrency in mind in designing computer systems is a must. Concurrency enables designers to modularize a system to carry out designs in a systematic and hierarchical manner. However, handling interactions between concurrent behaviors can be sometimes tedious, especially when the number of behaviors increases. For example, each behavior, created to achieve a specific goal of the system requirements, may conflict with other behaviors of the same system. Even if all behaviors can co-exist, the timing, i.e. the order of the behaviors to perform, may lead to unwanted results which violate the system requirements. Furthermore, behaviors created in the early design stages may not be realistic in the implementation stage. For example, a very fine grain of concurrency in software may introduce a heavy overhead of context switching which impacts the performance of the system. However, coarse-grain concurrency is often identified in the specifications (can be formal or informal). Dependencies may occur between behaviors so that in an extreme case every behavior is dependent on another in the system. Therefore real parallelism does not exist to assist the designer in exploring the maximum benefit of platforms such as multicore or distributed architectures.

Methodologies have been introduced to help designers to define and implement concurrency of computer systems in an appropriate manner. Related theories and concepts of these methodologies are detailed in the following sections.

To understand the methodologies of designing computer systems, one must know how computer systems are constructed. Their elements in computer systems can be categorized into software and hardware. Functionalities, according to their nature, are mapped to software or hardware taking into account several considerations such as performance and available resources.

2.2.1 Hardware in computer systems

Having hardware components in computer systems is obvious, since a computer is itself hardware. A hardware component can be implemented in analog or digital fashion. The latter is the focus in this thesis. Hardware components in computer systems can be ‘general purpose’ or ‘application specific’. General purpose hardware components are those common in most systems, and follow various interfacing standards so that they can be integrated with minimal effort. Application-specific hardware components, which perform required specific functionalities, are integrated according to the systems' needs.

Processors, sometimes called ‘processing elements’ (*PEs*), are examples of hardware components. Therefore PEs can be categorized as general purpose, such as embedded processors, and application specific, such as digital signal processing (DSP) processors; they differ by the computations performed. PEs can be facilitated with specialized functionalities, in the form of built-in hardware or ‘co-processors’, to enhance processing power, e.g. multicore architecture and floating point units. Directions of how computations are performed within processors are stored as software components, which will be detailed in the next section. Processors can be considered as the middle layer between other hardware and software components.

There are a number of ways to design hardware components. As technology advances, hardware design strategies evolve so that designers can describe hardware in less complex ways. Digital hardware systems are described nowadays using higher level languages called ‘hardware description languages’ or *HDLs*, such as VHDL [Lipsett et al., 1986][IEEE, 2000] and Verilog [IEEE, 2001]. Each hardware component has its dedicated behavior which can sometimes be further refined into sub-behaviors. A functionality of behaviors can be sequential or concurrent. In this thesis, behaviors and sub-behaviors are considered hierarchical and mostly concurrent, and are generally called ‘hardware process(es)’ in HDLs.

2.2.2 Software in computer systems

The cost of the development of embedded-systems software has an increasing trend with the evermore significant contribution to the total cost of system development [Allan et al. 2002]. Sequential behaviors are realized in software components, or software, and are executed by PEs. Software is described by using programming languages which can be high level such as Java [Arnold et al., 2000] and C [Ritchie et al., 1975] / C++ [Stroustrup, 2003], or low level such as assembly, in the form of ‘software source codes’. These latter are compiled into ‘software programs’ (or binaries) which are usually instructions of the PEs or virtual machines (e.g. JVM of JAVA). A software program that resides in storage such as hard-drives or memories can be accessed, loaded, and executed by the PE, as a ‘software process’, or just ‘process’ for short. The actual execution of a software process, in PE, can be out-of-order or parallel depending on features of the PEs. However, in this thesis, it is considered that a software process is executed sequentially, yet this does not stop multiple software processes being executed at the same time. Parallelism, or rather concurrency, of such processes is achieved via the use of operating systems (OSs).

2.2.3 Software concurrency and operating systems

Having multiple software processes running on the same processor is not new. Similar concepts appeared as early as the 50's when multiprogramming systems were developed [Rochester, 1955]. The software that coordinates multiple programs running together is known as an operating system (OS). OSs came with very basic functionalities in the earlier years, such as support for creation and deletion of software processes, and were later enhanced with other features to provide services, e.g. communication and synchronizations between software processes.

There are many flavors of OSs, characterized according to target applications requirements; for example timing constraint (required to finish a specific computation with given time, also known as 'real-time'), size (the available storage for both programs and the OS), and available executing platform (single or multiple processor architectures).

With respect to the timing aspect, OSs are differentiated by how software processes are dispatched by the schedulers. These follow different scheduling policies and implementation so that the OSs can be cooperative (processes release use of the processor voluntarily), pre-emptive (execution of software processes can be interrupted by the OS), and real-time (process executions are constrained by given times). Executions of some real-time systems are supported by real-time OS, or 'RTOS'. An OS scheduler can be equipped with more than one scheduling policy, such as earliest deadline first (EDF), rate monotonic (RM), round robin (RR), and cooperative scheduling, to achieve higher adaptivity.

In terms of size, 'embedded OSs' are used in the embedded applications in which storage is usually limited. To achieve a smaller size, these OSs can be modular (can be stripped down) and/or statically linked variants of general OSs, for example eCOS [Massa, 2003] (in relation to Linux).

The number of software processes running in a truly parallel manner depends on the target execution platforms on which suitable OSs are used. For single processor architecture, software processes are scheduled to obtain virtual parallelism to achieve better responsiveness and better use of inputs and outputs (I/Os). When multiple cores or processors are available, the underlying OS is then able to dispatch multiple processes running at the same time (true parallelism) to achieve performance gain. Communication and synchronization are also managed by the operating systems to handle dependencies of software processes running in parallel.

Even though OSs and the categories to which they belong are different, this does not infer that the types of OSs have to be exclusive. For instance, an embedded OS does not necessarily need to be a real-time OS. Similarly, a desktop OS, such as Linux, provides real-time scheduling mechanism when required by the applications.

Each software process can be further composed from a number of threads, which are concurrent. Threads are light-weight processes [Bovet et al., 2002] which usually share the same address spaces; i.e. they operate in the same area of memories, in contrast to processes which have their own memory spaces. Threads can be seen as a fine-grain of concurrency within coarse-grain concurrent software processes. Executions of threads are different from one implementation of OS to another. Threads can be implemented at kernel-level or user-level. The former are mapped to processes managed by the OS scheduler, while user-level threads are mapped to a single process whose internal scheduler is governed by specific libraries, or implemented by the designer. For instance, pthread [POSIX, 2009] is a library implemented by using kernel-level thread; on the other hand, GNU pth [Engelschall & Pth, 2006] operates at user level.

In some embedded OSs, such as MicroC/OS-II [Labrosse, 2002], the term ‘task’ is used to describe an execution entity. Tasks can be considered as either processes or threads, again depending on how the OSs are implemented, i.e. tasks to share a global address space or not. Throughout the thesis, the terms ‘process’ and ‘threads’ are used to differentiate the memory model of the concurrent execution entities.

Hence, important concepts to provide software concurrency by OSs are as follows:

1. Critical section (CS) - also known as a critical region where a process will not be interrupted when entering the CS.

2. Multitasking and scheduling - the scheduling is to provide concurrent process executions to prevent monopoly over, or starvation of, resources.
3. Context switching and interrupt – context is the snapshot of the processor state, which represents the status of the current executing processes. Context switching is required when process scheduling/switching occurs. Conventional processor provides an interrupt mechanism to store process context, where OS is responsible to arrange the location where the process context is saved.
4. Communication and synchronization – since processes are not only independent of each other but most often heavily interdependent, features like communication and synchronization are required.

Other OSs can be further application specific, such OSEK/VDK for automobiles [OSEK, 1997]. Further details of OSs can be found in classical texts such as [Silberschatz & Galvin, 1998].

Concurrent software behaviors may be sequentialized into a static single thread with dependencies between processes resolved in advance. One such approach is used in synchronous languages, e.g. ESTEREL [Berry & Gonthier, 1988]. In this case operating systems are not required for handling concurrency but may still be required for interactions with I/Os [Andre & Péraldi, 1993].

Software concurrency with the help of the OS plays an important role in this thesis. The libGALS and libDGALS, presented in Chapter 4 and 6 respectively, are libraries implemented using features (locking and scheduling) provided by the OS and benefiting from the multicore/multiprocessor architecture when supported by the OS.

2.3 Concurrency in system design

Designing systems start from specifications at a coarse-grain level of details, to a fine-grain level in implementation. A list of the characteristics of a system, e.g. how it behaves, is given in the specification of each system. Behaviors specified in this level of abstraction are not finalized and are implementation dependent. As an example, if two behaviors are concurrent, how they are scheduled is based on which scheduling policy to use, is unknown in the specification. Therefore, concurrency in the specification is ‘partially ordered’; that is, the order of the behaviors may be neither deterministic nor is final.

For example, in systems with a single processing unit, and making use of OSs, each behavior is ordered individually. The scheduling of behaviors sequentializes the activities of behaviors. Such sequences of how behaviors are scheduled may vary from one OS to another, or even in different scenarios with the same OS. In multicore systems, behaviors with dependencies still have a partially ordered relationship, while the independent computation will have no order at all.

Partial order of concurrency gives expressiveness to system specification, so that it is more flexible to laying out concurrent behaviors in a system under design. Behaviors at specification level can be later refined by following specific rules to achieve deterministic results. These rules, known as ‘model of computation’ (MoC), will be described in the later sections. Note that it is desired to still have non-deterministic concurrency in the implementation for the following reasons:

1. Limitation by the architecture: components in a system running in distributed networks act independently in general, and communicate with each other when required. These components do not share a global view of the system, to reduce unnecessary overheads in maintaining such a view.
2. To achieve dynamicity: a system may react to the environment or make a request to the environment to have behaviors activated at run-time. That is, the number of behaviors running at a given time is not fixed and not predictable. This enables systems to have both dynamicity and robustness.

The GALS-Designer framework detailed in Chapter 5, empowered by the libGALS library and SYSTEMC system level designing language (SLDL), provides the means of describing software concurrency in Globally Asynchronous Locally Synchronous (GALS) MoC (from libGALS) with the ability to specify partial concurrency (from SYSTEMC). The DynamicGALS framework in Chapter 6 further provides the ability to program distributed systems.

2.4 Model of computation, languages, and libraries

2.4.1 Model of computation

‘Model of computation’, or ‘computational model’, describes how behaviors are performed and how they communicate with each other in a system (composition of behaivors). Aspects of MoC include computational complexities, compatibilities, and language semantics. MoCs are not limited to being described in a purely mathematical manner. Various languages are proposed to work with dedicated MoCs to describe systems. ‘Formal languages’ are based on rigorous mathematical models, and therefore analysis can be made to explore their characteristics. On the other hand, ‘informal languages’ do not follow specific MoCs and extra effort is required to ensure the correctness of the designs. MoCs can be heterogeneous, i.e. merging concepts of various MoCs and presented in a unified view as in [Lee & Sangiovanni-Vincentelli, 1998].

2.4.2 Languages as design tools: concepts and backgrounds

Languages can also be ‘implemental’ or ‘theoretical’. Implemental languages, such as programming languages and hardware description languages, have compiler support to generate implementation in software or hardware from the source codes (or the source descriptions) of the design. In contrast, theoretical languages can only be expressed in a textual manner, but have a solid theoretical background to analyze the designed systems. Note that theoretical and implemental languages are not mutually exclusive; that is, compilers can be implemented for a theoretical language to make it an implemental one. Languages can be seen as tools to help system design, and are represented in many forms:

1. Mathematical formalism: alphabets (symbols) and strings of the language are defined, along with a set of the fundamental (kernel, or logical axioms) of the language. The fundamentals are further extended (or substituted) in a logical and mathematical manner to form a complete syntax of the language, for instance, functional programming languages, which are based on λ-calculus [Church, 1932]. Examples of such languages include ML (e.g.

STANDARDML [Milner, 1997]) and HASKELL [Jones, 2003]. Languages like ESTEREL [Berry & Gonthier, 1988], LUSTRE [Caspi et al., 1987], and SIGNAL [Benveniste et al., 1985] follow ‘synchronous formalism’, which is closely related to the GALS MoC used in this thesis.

2. Graphical representation: a set of graphical elements, such as nodes (vertices), arcs (edges), and labels, are used to construct a language. The rules of connecting these graphical elements are defined as the syntax of the language. Examples of languages with graphical representations are STATECHARTS [Harel, 1987], Kahn process network (KPN) [Kahn, 1974], and Petri nets [Petri, 1962].
3. Programming languages: can be general or application specific. General programming languages, such as C (which is an ‘informal language’), are suitable to describe systems in various application domains. Application-specific languages are designed for particular domains. For instance, synchronous language ESTEREL targets reactive systems. Compiler/translator application-specific languages may generate codes in general programming languages which often have portability in mind. Programming languages also come with different flavors, e.g. ‘imperative’ (closely related to state-based formalism, such as C/C++ and JAVA), ‘data-flow’ (used in signal processing, such as SIMULINK), and ‘functional’ (as afore- mentioned, e.g. HASKELL).

Languages can be presented in combinations of forms. For instance, SIGNAL is based on synchronous formalism, and can be presented in a graphical manner as data-flow, relational, and declarative [Le Guernic et al., 1991]. SIGNAL is implemental and its compiler generates codes in C, FORTRAN, and OCCAM [Benveniste & Berry, 1991].

With the help of ‘compilers’ and ‘translators’, the source code in one representation can be used to produce the resulting code in another. The differences between compilers and translators can be summarized in the following:

1. A compiler parses the source codes of a specific language, and uses an intermediate format to store the parsed result. The structure of the intermediate format is different from the source code, e.g. imperative style source codes are stored in a tree-structured control-flow representation. Instead of using the term 'compiler', 'synthesizer' is used in digital hardware development, which translates higher-level description of hardware to lower-level implementations on FPGAs/ASICs.
2. A translator provides direct mapping from the source language to the destination language. When direct mapping is not available, substitutions or macros from destination languages are used.

2.4.3 Library based approach

Software libraries, or 'libraries', are implemented at the top of the programming languages. Libraries take advantage of the existing language so that it is not necessary to design a new compiler. Run-time supports provided by libraries have more flexiblility than static checking in a language-based approach. It is also possible to bind (obtain help) with other libraries to merge different designing concepts. A library can be implemented according to a specific MoC or multiple MoCs. Even though libraries do not enforce designers to construct a correct program as a compiler does, they are still able to offer extra features, in terms of programming constructs, to reduce designers' efforts in describing systems in raw source codes which are error prone.

2.4.4 Current state of the art and approaches

MoCs, languages, and libraries have been proposed and developed to cope with concurrency in designing hardware, software, and overall systems. Some examples of MoCs and corresponding developments are as follows:

1. Discrete event (DE): this is generally used in hardware description languages (HDLs), and will be detailed in Section 2.5.
2. System-level design languages (SLDLs): these are dicussed in Section 2.6 and can be used to describe systems in different levels of abstractions. It is also possible to generate software and hardware from SLDLs.

3. Process calculi and process networks: these are used to describe relationships between concurrent processes and will be described in Section 2.7.
4. Actor-based models: that describe autonomous concurrent entities and the interactions between them. They may or may not follow a MoC but are widely used in different fields, as described in Section 2.8.
5. General programming languages with support to describe concurrent processes: these are made as built-in constructs to the language itself, or libraries of existing languages to provide concurrency. Some of the programming languages borrow the concepts of other MoCs as part of their features. See Section 2.9.
6. Synchronous and reactive MoCs (S/R): these target reactive and time-critical systems. Determinism is a key factor of these MoCs. Extensions and relaxations to them have been proposed for wider uses. S/R MoCs and developments are detailed in Section 2.10.
7. Globally asynchronous locally synchronous (GALS): this is used in both hardware and software domains. GALS can be seen as a close relative to the S/R MoC, and will be presented in Section 2.11.
8. Dynamic GALS (DGALS): this is a newly proposed MoC which merges concepts from the Actor-based model and the GALS model. A brief description of DGALS is in Section 2.12.

In this thesis, libGALS and libDGALS are libraries implemented by following GALS and Dynamic GALS (DGALS) MoCs to enrich general programming language (in this case, C).

2.4.5 Synchronous versus asynchronous

The terms ‘synchronous’ and ‘asynchronous’ are used widely in the field of designing computer systems and MoCs. In this section, the terms are further described and are used throughout the thesis to prevent ambiguity. These terms have been adopted in various scenarios:

1. How concurrent behaviors are carried out: e.g. synchronous concurrency and asynchronous concurrency. In synchronous concurrency, concurrent behaviors follow the same logical time reference, similar to clocks in digital hardware design. Synchronous languages are developed to target synchronous concurrency and are detailed in Section 2.10; asynchronous concurrency is more general. Most of operating systems, programming languages, and libraries which offer process creations and support threading follow the model of asynchronous concurrency. In this thesis, synchronous and asynchronous are used to present concurrency in MoCs.
2. How communications are made: e.g. synchronous send-and-receive in contrast to asynchronous send-and-receive. Synchronous communication can sometimes be referred to rendezvous, where both the sender and receiver are blocked until the ealier communication has been completed. Asynchronous communication, which incorporates the uses of buffers, may not stop (block) the sender and receiver during the communication. There are variants of asynchronous communication; for example the sender and receiver may be blocked when the buffer is full and empty respectively. The 'send-and-forget' model does not block the sender at all, nor does guarantee that the data sent will be received.
3. How function calls are issued: e.g. synchronous function call and asynchronous function call. Synchronous function calls will block the execution of the caller until the results of the call are returned. General programming languages implement synchronous function calls to immediately evaluate the outcome so that the next operations, which may rely on the outcome, can proceed. On the other hand, the caller continues to run after issuing the asynchronous function call. The caller may be blocked in the future when the return value of the function call is required. Asynchronous function calls are often adopted in the distributed computing environment. An asynchronous function call is close to the concepts of 'future and promises' [Liskov & Shrira, 1988]. The difference between

(a)synchronous communication and function calls is that communications are performed by both parties without a given order and can be active, while the called party (the function or the required service) is passive.

Models of concurrency, communication, and function calls may not be directly related. For example, in synchronous computation, communications are achieved through signal/event broadcasting, which is neither synchronous nor asynchronous, but governed by the MoC so that the dependencies occurring between communications are resolved.

2.5 The discrete event MoC and HDLs

Hardware description languages are used to specify digital hardware at a higher level to reduce the effort of designers in constructing large digital systems. Higher level descriptions are synthesized to lower-level logic and bit-streams that will be used to create the actual design on digital hardware including FPGA and ASIC. The level of describing hardware components depends on the requirement of the design stages. This enables designers to have abstract views of the system before implementing them fully, although not every model described at a higher level is synthesizable.

Verilog [IEEE, 2001] and VHDL [Lipsett et al., 1986] are well known HDLs and standards in the industry. The discrete event (DE) MoC is adopted by HDLs, in which concurrent behaviors of hardware processes are represented as events and governed according to the DE MoC.

The events are chronometric [Le Guernic et al., 2003], which means that the time of the occurrence is attached to each event. Events are queued upon on their generation, and are dispatched by the simulation kernel which can be made generically or to specified target hardware. The simulation kernel has a sense of time steps. The simulation kernel scans through the event queue to dispatch the events whose time of occurrence matches the time steps. Further events can be populated by the dispatched events. The dispatcher of the simulation kernel scans the queue until there is no event of the current time step left or populated, when the simulation kernel will carry out the next time step.

DE MoC provides a way to handle concurrency. However, interleaving execution of behaviors leads to non-determinism [Benveniste et al., 2003]. This makes DE sometimes not suitable for modeling of critical systems.

2.6 System-level design languages

2.6.1 The need for system-level design languages

Complexities of a system can be due to interactions between behaviors and how to implement behaviors in HW/SW components (e.g. new designs or existing intellectual properties, IP). Other important factors in designing a system include constraints such as the availability of resources. With the growth of design complexity of computer systems, various approaches are proposed to increase the designers' productivity and shorten the time and effort between the specification and implementation of such systems.

Programming languages such as C/C++ and Java are also used for specification due to their flexibility in describing functionalities, their data abstraction abilities, and their huge support in the form of software libraries. At the early design stage, components (both hardware and software) of the final system implementation might not be identified without taking consideration of different aspects such as performance evaluation. Similarly, hardware description languages lack the support of describing software components of the system. It is not easy to model software concepts such as data structure and algorithms that include recursive functions in HDLS.

Single language specification is also a need in conquering the system design [Lavagno & Sentovich, 1999]. This leads to the requirement for a language to bridge the design in software and hardware, as well as to have a higher level of abstraction; in design, in this case, system-level design languages (SLDLs) are proposed.

2.6.2 System-level design languages based on existing languages

SLDLs such as SYSTEMC [OSC Initiative, 1999] and SPECC [Gajski et al., 2000] are proposed. SystemC is a library extension to C++, which provides a set of classes and macros to empower the developer with the mechanisms to describe hardware (close to what hardware description languages do) and software systems in a single model. Similarly, SPECC facilitates constructs to describe systems at an earlier design phase for both specification and system level synthesis. Differences between SYSTEMC and SPECC are described in [Cai et al., 2003]. SLDLs come in different flavors, unlike SYSTEMC and SPECC which are based on imperative languages; BLUESPEC is based on Haskell and can provide different levels of abstraction.

System-level languages such as SYSTEMC and its simulation kernel follow DE MoC, and hence it is possible to have non-deterministic behaviors between simulation runs. However, as mentioned previously, such non-determinism also allows the model to be described in a more flexible manner as a trade-off.

Other tools in industry adapt the single language approach to system design, such as Synphony C from Synopsys. Synphony C is based on C/C++, by use of which descriptions of the system are made and are compiled via the Synphony C compiler to generate hardware in RTL and software in C.

System-level design, which relies on SLDLs, is carried out with various proposed methodologies, detailed in Chapter 3. Descriptions of SLDLs in higher-level abstractions are further refined, manually and/or automatically, towards the implementation. Because SLDLs are based on existing programming languages, interfacing between existing software libraries and other programs is viable. Simulation approaches, which make use of simulation kernels of SLDLs and other existing simulators, are proposed. Some commercial simulators, such as ModelSim, have the ability to perform mixed language simulations, by having components modeled in different languages such as SLDLs and HDLs.

2.7 Process calculi and process networks

Describing concurrency in a mathematical fashion has been developed. CSP [Hoare, 1978] and CCS [Milner et al., 1980] are two of the most notable examples of process calculi, which express how concurrent processes (in an abstract form, can be hardware or software) evolve. Communication between processes is also presented in process calculi, such as rendezvous in CSP. Process calculi are primitive and insufficient for an implemental language. However, they are often implemented as features in programming languages.

Process networks are also used to address the concurrency and interactions of processes/behaviors (again, does not have to be a hardware or software process). Petri-Net (PN) [Petri, 1962] and Kahn process networks (KPN) [Kahn, 1974] are examples of such networks. Both PN and KPN are presented graphically. PN is described as a composition of places (conditions) and transitions (or events, which are concurrent processes). KPN is presented as concurrent processes which produce and consume tokens to/from the unbounded FIFO buffer inbetween. PN is used to described control (can be used to describe data) and can be non-deterministic, while KPN is for data and is deterministic. Despite the difference in how concurrent processes are described using PN and KPN, these process networks are based on a concept: tokens are generated by the producers, and when enough tokens have been gathered (conditions fulfilled), the consumer of the token will proceed (or fire).

Restrictions are made on these process networks so that they can be implemented. For example, PN can be restricted and converted to FSM for deterministic analysis [Peterson, 1977]. Statically schedulable data-flow (SSDF) [Lee & Neuendorffer, 2005], previously SDF (Synchronous data-flow) [Lee & Messerschmitt, 1987], restricts the size of the FIFO buffer in KPN so that the rate of the processes can be solved as linear equations.

Both PN and KPN are graphical formalism, which is intuitive. However it is difficult to manage for large scale programs [Jose et al., 2009]. Programming languages such as LUSTRE [Halbwachs et al., 1991], Synchronous KPN [Caspi & Pouzet, 1996], and libraries such as NRP [Boussinot, 1992] are proposed and are inspired by the KPN with the concepts of synchrony.

2.8 Languages based on Actor-based models

The Actor model was proposed in the 80s as another model of concurrency [Agha, 1985]. Actors perform at their own rate, and communicate asynchronously with each other through sending messages that are buffered in mailboxes [Boussinot et al., 1996]. Mobile agent platforms such as JADE [Bellifemine et al., 2005], enable agents to operate in an autonomous manner, with the ability of migration, similar to Actor-based models. The Actor model has been implemented as programming languages, and libraries supporting the operations of actors. Axum [Microsoft Corporation, 2008], based on the Actor MoC, is a programming language as a part of the .Net framework. Active Object [Lavender & Schmidt, 1995] implements the Actor model using C++. Detailed comparisons of various related models can be found in [Nikaein, 1999]. MoC related to the Actor-based model offers the following:

1. Asynchronous executions of actors. Each actor generally operates independently.
2. Asynchronous and synchronous communications. There are many choices of communication models for an actor to choose from.
3. Mobility of actors. An extension of the Actor model, actors/agents are able to migrate to the required computation node to perform actions. Actors/agents which are able to migrate are called mobile agents.

2.9 Programming languages with concurrency

General programming languages like C, C++, and Java are used to describe transformational systems. Algorithms which are computational behaviors are specified using general programming languages. Concurrent behaviors of transformational systems are supported by the built-in constructs of the programming languages or other means such as uses of (real-time) operating systems. Processes/threads are used to represent the corresponding concurrent behaviors of the software. For example, user-typed class implements Java Runnable class will be viewed as a thread to the underlying Java program. Similarly, threads or processes (of programs) can be implemented in C and are governed by the operating systems. Communications between concurrent threads and processes are achieved using shared variable or inter-process communications (IPC), which can be both asynchronous and synchronous.

Programming languages are implemented based on different conceptual models. As an example, general programming languages (C/C++/Java) follows the implicit 'state-based imperative' style. Functional languages, such as Haskell and ML, are influenced heavily by the λ-calculus. The concurrency provided by programming languages does not necessarily follow any aforementioned model of concurrency; some may follow but may not be restricted. Threads created using pthread library (or user threads available in the ML) do not follow any MoC, and are controlled by the operating systems. This creates a scenario that even if the program is correct, i.e. a race condition never happens, the execution outcome may differ due to the scheduling policy which might be affected by the load of the machine at various times. In this case, MoCs are enforced through programmers' efforts or the uses of libraries that provide programming interfaces to ease the load of the designer.

Some programming languages are built on top of existing ones through adding constructs of concurrency, which introduce new syntax to the base language, to support the desired MoCs. Compilers then map the introduced construct to codes in the base language or to other languages. For instance, Scala [Odersky et al., 2004] which provides the flavor of functional programming based on imperative Java language, which supports concurrency in the Actor model and CSP. In contrast, Erlang [Armstrong et al., 1993], which is also based on the Actor model and CSP MoC, is not based on any language.

2.10 Synchronous MoC and approaches

2.10.1 Introduction of synchronous and reactive programming

Non-determinism, which can be observed in concurrent software, can be caused by temporal logics [Berry & Gonthier, 1988] and race conditions [Lee, 2006], which are introduced with uses of operating systems. Implementations of synchronous languages do not rely on conventional mechanisms such as operating systems, but respect the ‘synchrony hypothesis’ to ensure determinism. Software development benefits from synchrony hypothesis, implemented through uses of synchronous languages, are summarized in [Benveniste & Berry, 1991], [André, 1996], [Halbwachs, 1998], [Benveniste et al., 2003], and [Potop-Butucaru et al., 2005].

In synchrony hypothesis, input events are gathered at the beginning of each tick of logical time and corresponding outputs are generated in ‘zero-time’. Concurrent behaviors of synchronous systems are carried out in a number of discrete steps, called reactions, instants, or ticks. Barrier synchronizations are exercised by each concurrent behavior at every tick. Communications between behaviors are via ‘signal (or event) broadcastings’. Pre-emption is one of the key control mechanisms within concurrent behaviors. Throughout this thesis, tick, signal, and pre-emption will be used as the major terminologies with respect to the synchronous languages and S/R MoC. Mealy machine and digital circuits generated from synchronous languages are based on mathematical models which are deterministic and can be verified by using the technique described in [Clarke, 1997].

Synchronous languages ESTEREL [Berry & Cosserat, 1984], LUSTRE [Halbwachs et al., 1986], and SIGNAL [Benveniste et al., 1985] are the classical synchronous languages that were built in the styles of imperative, data-flow, and relational languages respectively. They are proposed to target the real-time systems by applying synchronous hypothesis. Such a concept is closely related to that of reactive systems [Harel & Pneuli, 1985] to design systems with real-time characteristics.

Reactive languages, closely related to synchronous languages, relax the synchrony hypothesis so that the absence of signals/events is known at the next instant/tick [Boussinot & Dabrowski, 2006]. Reactive approaches further enhance synchronous languages with the ability to create concurrent behaviors at run-time and to enable distributed reactive systems dynamically.

A comprehensive, but in complete list of other synchronous/reactive families includes: ATOM [Hawkins, 2011]**,** ARGOS [Maraninchi, 1991], Distributed reactive machines (DRM) [Susini et al., 1998], FAIRTHREADS [Boussinot, 2002], FUNLOFT [Boussinot & Dabrowski, 2007], ICOBJ [Boussinot, 1996], JUNIOR [Hazard et al., 1999], LOFT [Boussinot, 2005], Lucid Synchrone [Caspi et al., 2007], Nets of reactive processes (NRP) RC [Boussinot, 1992], POR (Programming of reactive object) [Doumenc & Boussinot, 1991], QUARTZ [Schneider, 2009], RAMA (Reactive Autonomous Mobile Agent) [Nikaein, 1999], REACTIVE C (RC) [Boussinot, 1991], REACTIVE ML [Mandel & Pouzet, 2005], Reactive Object Model [Boussinot et al., 1996], REACTIVE SCRIPTS [Boussinot & Hazard, 1996], REACTIVE SML [Pucella, 1998], REJO/ROS (Reactive Java Object) [Acosta-Bermejo, 1999], SL [Boussinot & De Simone, 1996], SUGARCUBES [Boussinot & Susini, 1997], and SYNCCHARTS [André, 1995].

2.10.2 SW and HW implementations of S/R approaches

Software implementation of synchronous languages can be categorized by how synchronous programs result from the original system descriptions in synchronous languages: (1) language-based, and (2) library-based. In language-based approaches, synchronous descriptions are compiled into several intermediate representations which are used to generate software source codes in host languages such as C, or directly to the platform assembly or machine codes. Host language source codes are then compiled into synchronous programs through use of the compiler for the target platform. On the other hand, library-based approaches are supported by the available primitive constructs provided by existing programming languages, and are added as the extensions to these languages in the form of function calls (or macros) as interfaces. Synchronous descriptions using these interfaces are compiled and are linked with the library to produce the target binaries.

As an example of compiler-based language, ESTEREL has a number of developed compilers with different compilation techniques such as ESTEREL v3 compiler [Berry & Gonthier, 1988], v4 [Berry, 1999], v5 [Berry, 2000], Columbia Esterel Compiler (CEC) [Edwards, 2002], SAXO-RT [Closse et al., 2002], and Potop-Butucaru’s compiler [Potop-Butucaru & De Simone, 2003], to sequentialize the concurrent behaviors of ESTEREL description into a static single-thread program.

Examples of library-based approaches are REACTIVE C, JUNIOR, and SUGERCUBES. REACTIVE C provides extensions to C to model synchrony. Concurrency within

REACTIVE C is sequentialized in the textual order of the system under design [Boussinot, 1991]. SL can be firstly translated to REACTIVE C then translated to C. Similarly, JUNIOR extends Java with dedicated JUNIOR kernel to support reactivity and synchrony. SUGARCUBES is close to JUNIOR, comprising a set of JAVA class to program synchronous reactive systems in JAVA.

Implementing graphical synchronous languages via intermediate representations or through extensions to existing programming languages exercises the mixture of compiler-/translator- and library- based approaches. For example, SYNCCHARTS is translated to ESTEREL [André, 2003] and then compiled into C [von Hanxleden, 2009][Traulsen et al., 2011]. ICOBJ is implemented based on SUGARCUBES/JUNIOR and REACTIVE SCRIPTS.

In order to execute synchronous software programs more efficiently, hardware enhancements and specialized processors are proposed such as REFLIX [Salcic et al., 2004], REMIC [Salcic et al., 2005], EMPEROR [Dayaratne, 2004][Yoong et al., 2006], KEP3a [Li et al., 2006], BAL virtual machine [Plummer et al., 2006][Edwards & Zeng, 2007], and STARPro [Yuan et al., 2009] for executing ESTEREL. Entities described in synchronous languages can be compiled into digital circuits based on techniques presented in [Berry, 1992], [Berry, 1999], [Malik, 1994], [Shiple et al., 1996], [Schneider, 2000], and [Edwards, 2003]. Hardware and software co-synthesis of ESTEREL also exists such as [Gädtke et al., 2007] where hardware implementation of synchronous reactions communicate with software implementing counterparts executed in the KEP processor.

The concepts of synchrony and reactivity have been used in fields such as multimedia and graphical system design. Examples include: Audio language CHUCK [Wang et al., 2003], Reactive animation [Efroni et al., 2005] with frontend of Flash and backend of RHAPSODY [Gery et al., 2002] which is based on STATECHARTS.

2.10.3 Other related approaches

ECL [Lavagno & Sentovich, 1999] and JESTER [Antonotti et al., 2000] are ESTEREL–like extensions to C and Java, respectively. Use of translators is adopted to separate the computational and the reactive parts to C/Java and ESTEREL. Single language approaches help the designer to concentrate on programming instead of interfacing components in different languages. However, debugging process on reactive parts will be working on the generated ESTEREL codes which may not be easy. Proposed support of asynchrony is via support of RTOSs and POLIS [Balarin et al., 1997], respectively.

Because synchronous languages can be used to describe software and hardware, using synchronous language as the backbone of the system-design framework has been developed. For instance, a system-design framework Polychrony [Le Guernic et al., 2003] is based on the multiclock feature of the synchronous language SIGNAL.

Systems which have components running at different clock speeds, such as distributed systems, are also addressed in the research community. Synchronous programs running on distributed network communicate with weak synchrony in CoReA [Boniol & Adelantado, 1993], that is, communications via signals are delayed for one instant, so that the overall program can be analyzed. A de-synchronization of synchronous programs in OC (object code) format with uses of FIFO buffers is presented in [Caspi & Girault, 1995]. De-synchronized program will be divided into distributed components. The overall behavior of the distributed program is the same as the original. Further discussions on distributing synchronous programs are detailed in [Girault, 2005].

Other reactive approaches which do not follow synchronous MoC exist. Reactive Java [Passerone et al., 1998] and Triveni [Colby et al., 1998] provide support to program reactive systems. However, without enforcement of the synchronous MoC, the designs will suffer in the same way as the conventional thread-based programs. SML (state machine language) [Browne & Clarke, 1985] and CSML (compositional SML) [Clarke Jr et al., 1991] are based on FSM to support reactive software and hardware; however, the ability of handle data computation is absent.

2.11 The GALS MoC and related developments

2.11.1 The concept of GALS

Globally asynchronous and locally synchronous (GALS) MoC have been proposed in [Chapiro, 1984]. In the sophisticated computer systems, or heterogeneous systems, there can be a number of processing units, such as processors integrated and interacting with each other. For instance, System-on-a-Chip (SoC) consists of processors running at different speeds of computation and communication [Potop-Butucaru & Caillaud, 2007]. To achieve global synchrony is impractical because the fast processor will have to wait for the slower one to achieve barrier synchronization.

The concept of GALS is originally incorporated for use in hardware design. The complexity and size of chip increases along with the operational frequency and introduces problems such as higher power consumptions and clock skew of single clock domain digital hardware. A GALS digital system is composed of different sub-systems (clock domains) which are running at their own speeds. Examples of communication and synchronization between sub-systems include stretched clocks, uses of FIFO buffer, and a specialized synchronization mechanism, which are discussed in [Krstić et al., 2007].

As mentioned in Section 2.4.5, the terms synchronous and asynchronous have been used in different contexts and with different meanings, and hence there are variants of GALS definitions. The concept of GALS in TinyGALS [Cheong et al., 2003], is based on the concepts of asynchronous and synchronous function-calls. Function calls at a global level in TinyGALS are performed through asynchronous message passing, while intra-component communications are through synchronous function calls as in programming languages. X10 [Charles et al., 2005], a distributed programming language, follows the same GALS strategy as in TinyGALS.

In this thesis, the definition of GALS is based on the co-existence of synchronous and asynchronous concurrency. The communication between asynchronous entities may or may not follow a specific model of communication. Asynchronous communications in GALS systems follow a deterministic model, such as CSP rendezvous, which can be analyzed along with each synchronous compartment, as the key benefit of using the GALS MoC.

2.11.2 GALS in the software domain

The synchronous subsets of the systems benefit from the existing synchronous languages emphasizing determinism. On the other hand, an asynchronous model is suitable for distributed networks [Berry & Sentovich, 2000]. Languages and compilers following the GALS concept have been utilized in software domain.

Language approaches to describe GALS systems are extensions of existing synchronous languages, such as Communicating Reactive Process (CRP) [Berry et al., 1993], Communicating Reactive State Machines (CRSM) [Ramesh, 1998], Multiclock Esterel (MCEsterel) [Rajan & Shyamasundar, 2000], or as a new languages such as SHIM [Edwards & Tardieu, 2006] and SystemJ [Malik et al. 2010].

SHIM [Edwards & Tardieu, 2006] is proposed to program asynchronous systems in which Khan network's channels with CSP rendezvous are used. The compilation process of SHIM ensures a single writer to a variable at a time to prevent data races. Synchronous systems can also be modeled with SHIM with suggested approaches in [Edwards & Tardieu, 2006]**.**

SystemJ [Malik et al. 2010] merges ESTEREL for synchrony and reactivity, CSP for asynchronous communication, and JAVA for data computations as a whole. SystemJ does not rely on the existing ESTEREL compiler, as ECL and JESTER do, and enriches the Java language with programming constructs to design GALS systems. Synchronous concurrency in SystemJ is described through reactions within clock domains, where they are asynchronous. Communication between asynchronous clock domains is through point-to-point channels following CSP rendezvous. As a language-based approach, a SystemJ program that is correct with regard to a specification will also be compiled to a correct implementation.

2.11.3 System-level design based on GALS

GALS approaches are also adopted in system-level design. POLIS [Balarin et al., 1997] has been developed as a HW-SW co-design framework. The framework is composed of CFSMs, co-design finite state machines, which are synchronous entities. Thus each CFSM can be translated into synchronous languages, in this case, ESTEREL, and can be verified [Berry & Sentovich, 2000]. CFSMs are connected to an asynchronous network, which categorizes POLIS as a member of the GALS family.

DFCharts [Radojevic et al., 2006] merges the concepts of SDF and hierarchical FSM to have the capability of designing control- and data-dominated systems. DFCharts adopts the GALS MoC where communication of asynchronous elements in DFCharts follows the CSP rendezvous. Descriptions made in SLDL such as SYSTEMC and synchronous language ESTEREL can be mapped to DFCharts [Radojevic et al., 2006] which is formal and intuitive.

2.12 Dynamic GALS MoC

Dynamic GALS MoC, as a further extension to the GALS MoC, incorporates the concept of pi-calculus [Milner, 1999], that is, behaviors are able to migrate from one computational node to another, similar to mobile agents. ULM [Boudol, 2004] presents a programming model to describe GALS systems with mobility in theory. Dynamic Synchronous Language (DSL) [Attar et al., 2011] is proposed based on the existing reactive approaches such as SugarCubes, ReactiveML, and FunLOFT. Synchronous behaviors can be dynamically created on distributed sites. However, communication between behaviors of different sites is not clearly defined.

DSystemJ [Malik et al., 2010] applies to the concept of dynamic systems which introduce process mobility to SystemJ, so that asynchronous clock domains and channels can be created at different computational nodes at run-time. In contrast to DSL, the formal semantics of clock domain migration and channel communications are given. The DSystemJ is followed to a large extent in this thesis when specifying dynamic GALS systems and libDGALS library in Chapter 6.

2.13 The library-based GALS/DGALS frameworks

Figure 2.1 illustrates the relationships between the MoCs (in rounded rectangles) and examples of related approaches (in ellipses). The DGALS MoC, which is surrounded by the related MoCs, particularly GALS MoC, and its use in supporting standard programming language C in this case, will be the focus in this thesis.

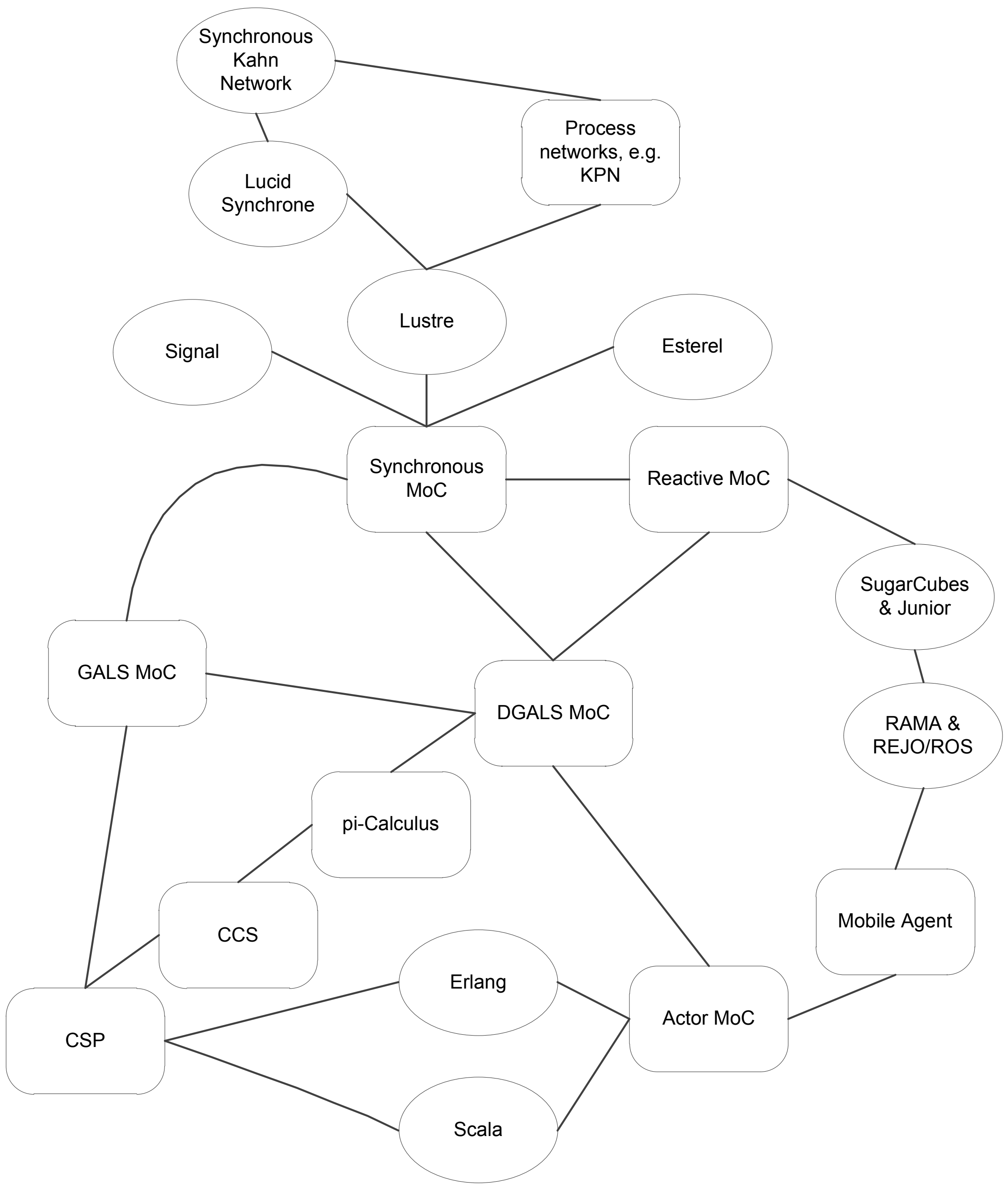


Figure 2.1: Relationships between MoC and approaches

In the next chapter, the SYSTEMC SLDL is used to model software concurrency by incorporating models of operating systems and software processes. The model of the OS consists of services to support general asynchronous concurrency and communication, as well as the dedicated service to support synchronous concurrency in the synchronous language. libGALS, a library-based approach that can be used to both describe and realize GALS systems in C is then introduced in Chapter 4. As a further development, libGALS is merged with SYSTEMC to enable modeling of entire systems that include both models of hardware and GALS software, and this is presented in Chapter 5. This enables the design of GALS systems in SYSTEMC. Finally, libGALS are extended with features of dynamic creation, termination and migration of asynchronous behaviors into the DynamicGALS framework, which enables the design of dynamic GALS systems. The approach is detailed in Chapter 6.

3
System design with OS modeling

A large number of computer systems have software implementation of concurrency with support from operating systems (OS). Many of those systems do not require full OS, but a reduced functionality that can be implemented in software, hardware or their combination. In order to model such systems it is not only necessary to provide OS functionality, but also the mechanisms to support software concurrency as well as interactions with hardware. Such a model is required to be suitable in different levels of abstraction in the early phase of design to explore the suitability of hardware/software partitioning and implementation. This chapter presents a methodology of modeling complete computer systems that include OS with basic functionality and extensions to ensure safe concurrency as the center of the system model. The approach is illustrated in comprehensive example.

The proposed modeling and design framework enables embedded software, which includes software processes and the OS, and hardware components, to be described and simulated together. This methodology, described in this chapter, also provides anexploration of features of the OS. The model can be further mapped on standard/customized OSs whose performance can be evaluated. Hardware/software implementation can be achieved according to such evaluations which determine the trade-off option. The proposed model is based on the use of SYSTEMC as its backbone. But the methodology allows the inclusion of models developed in other languages. For example, hardware components may have already been developed in hardware description languages (HDLs).

This chapter is organized as follows: in Section 3.1 approaches to system level design are discussed. This is followed by detailing the design stages in Section 3.2. Section 3.3 introduces the concept and existing approaches of OS modeling in system design, as well as the hardware support for OSs. It is followed by the proposed system model with OS modeling detailed in Section 3.4. The modeling of OS and software processes is introduced in Section 3.5 and 3.6, respectively. A framework adapting the uses of OS and the processes model to explore the possibilities of customization of OS is presented in Section 3.7. A case study where an application originally modeled in ESTEREL is mapped on the SYSTEMC based new framework is described and analyzed in Section 3.8.

3.1 Approaches to staged system level design

Approaches in system-level design are iterative, a step-based design with feedbacks from each step being taken and refinements made from the feedbacks. Iterative steps are carried out at higher levels of abstraction, to prevent unnecessary effort on details at lower abstraction levels. Design ideas, performance evaluation, architectural feasibility, component selections, and system integrations are taken into account to give feedback for the refinements. Figure 3.1 illustrates how system-level design is carried out. The horizontal axis of Figure 3.1 represents the design stages of the earliest specification-capturing at the beginning, which is at the left end of the axis. Levels of abstraction used in the design, from the most abstract, such as untimed functional, to the most detailed cycle-accurate level, are represented by the vertical axis. System-level design methods can be categorized into three groups according to how the overall system is constructed, and are described in [Cesário et al., 2002], and later in [Cai et al., 2003]. They are identified as ‘system-level synthesis’, ‘component-based design’, and ‘platform-based design’.

Entry points of these approaches, shown in italics in Figure 3.1, demonstrate the relative timeline and level of abstraction where these approaches are carried out. For instance, system-level synthesis starts from the top-left corner of Figure 3.1 and illustrates such approach starts at the highest abstraction, i.e. untimed function, while component-based design performs the selection of existing components, which are modeled or implemented in the cycle-accurate fashion.

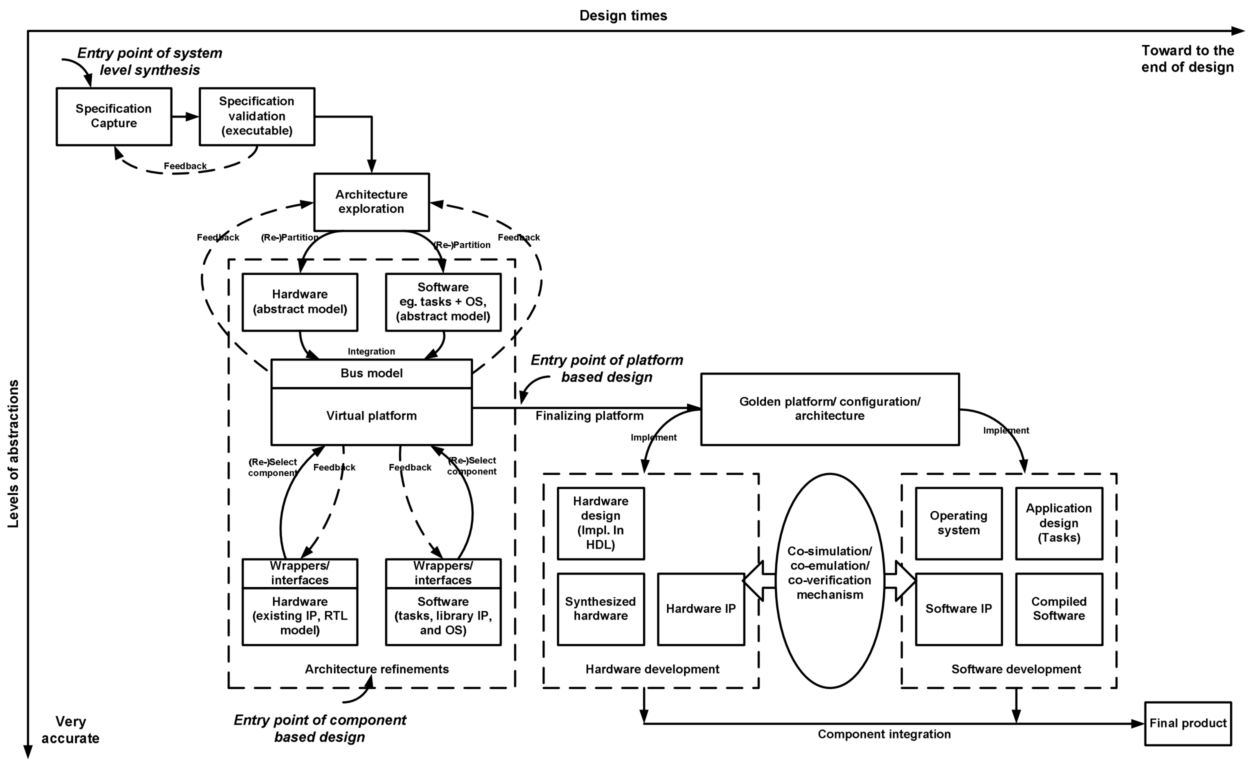


Figure 3.1: Staged system-level design

3.1.1 System-level synthesis

System-level synthesis follows a top-down approach, where implementation details are not known and will be derived from the specification of system behavior. During the refinement process of the system specification, software/hardware portioning is performed, followed by the software and hardware synthesis at the end. This methodology is adopted to explore the best configuration of the system components possible.

Specification of a system is first made informally, then transformed to a more formal representation/model, resulting in an executable on the host machine when using SLDL such as SYSTEMC. Behaviors of the systems are identified at this stage. This procedure is made at the earliest design stage shown as the entry point of system-level synthesis in Figure 3.1. The executable is used to validate the correctness of the model with the given specification. The execution model can be used to validate both the implementation of the final design and identified functional specification at the beginning of the design phases. Feedbacks are given to correct the modeled design, or to report if the specification is not feasible. Once the end of feedback-refinement iterations is reached, the ‘architecture exploration’ will be performed.

During architecture exploration, behaviors in specification are mapped to hardware and software components, known as hardware/software (HW/SW) partitioning in ‘architecture refinement’, according to characteristics of behaviors and constraints such as available resources. A number of ‘virtual platforms’ are obtained in architecture refinements performed iteratively, similar to specification validation. The hardware model at a higher level of abstraction, and the software model comprising processes and the OS model are integrated and communicate with each other through a bus. Note that the models of hardware and software are still abstract and can be replaced interchangeably. It is also possible to use implemented components in the architecture exploration. Interfaces will be required to adapt the uses of existing components. In this stage, communication and computation are modeled in various levels of abstraction, which provide more information to the designers towards the final architecture/platform. Information such as timing is obtained through various approaches and added (annotated) to the virtual platform to evaluate overall performance as feedbacks for better partitioning.

At the end of the refinement iterations, the final platform is determined. Such an optimal platform is also known as the ‘golden model’ [Black et al., 2008] or ‘golden architecture’ [Cesário et al., 2002]. Because hardware and software development of the golden platform start simultaneously and are generally carried out at different speeds, techniques to use models at different levels of abstractions are adopted again in hardware/software co-simulations to test the unit under design, which can be either software or hardware components. Various co-simulation techniques have been introduced and investigated. Finally the verified hardware and software are merged to the final product as the end of design.

3.1.2 Component-based design

Component-based design [Cesário et al., 2002] is a bottom-up strategy in which a platform is constructed with interconnecting available components. An entry point of the component-based design is shown in Figure 3.1. Existing components are used to construct the virtual platform and feedbacks are given to perform re-selection on components to establish the golden platform. Components can be hardware and software IPs. Interconnects between hardware IPs, also called buses, can either be selected from available implementations, or generated as wrappers. Similarly, the OS that manages the software processes, is selected, or generated as software wrappers. Once the golden model is formed, the development will be carried out as the system-level synthesis.

3.1.3 Platform-based design

Platform-based design [Sangiovanni-Vincentelli & Martin, 2002] is considered a special case of the top-down design approach [Cai et al., 2003]. It is also a special case of component-based design where hardware platforms (sets of components) may be pre-determined. In this case, software development may be based on the existing libraries. Generally, the skeleton of the platform, for instance the hardware bus, is predetermined. As illustrated in Figure 3.1, the entry point of platform-based design is close to the golden model. The platform can be customized by selecting suitable hardware components. Different sets of configurations of platforms are called ‘platform instances’. Standardizing interfaces such as PCI/Express provides connectivity to other components facilitating video and audio features of the system. Platform instances of each desktop computer may differ and the optimal platform depends on the target usage of such systems, e.g. graphical design or networking servers.

3.2 Stages in system-level design

Different levels of abstraction are based on the degree of accuracy of the underlying model. Various levels of precision have been used in different dimensions when applying different accuracy on the system models. These dimensions include data granularity and timing in communication [Ghenassia, 2005], timing computation (functionality) and communication [Cai et al., 2003], and abstractions of interfaces for co-simulation [Yoo & Jerraya, 2005]. Aspects of modeling at different levels of accuracy, from the most abstract to detailed, are listed as follows:

1. Data granularities in communication: application packet, bus packet, and bus size [Ghenassia, 2005].
2. Timing accuracy in communication: untimed, approximately-timed, cycle-accurate [Cai et al., 2003] and [Ghenassia, 2005].
3. Timing accuracy in computation follows the preceding case [Cai et al., 2003].
4. Hardware interfaces in different abstraction: cycle accurate, transfer level, transaction level, and message level [Yoo & Jerraya, 2005].
5. Software interfaces: instruction set architecture (ISA) level, device-driver level, and OS level [Yoo & Jerraya, 2005].

Modeling approaches are based on two major properties: communication and computation. Communication specifies how one component interacts with others. Computation specifies the way an algorithm is carried out in a software and hardware IP. Both communication and computation comprise co-relevant features: timing accuracy and data granularity.

In hardware/software co-design, communication is modeled between 1) hardware components, 2) software components, and 3) software and hardware components. Timing accuracy in modeled communication is categorized as follows:

1. Untimed: no timing information is in the model.
2. Time approximate: timing information is included. Timing information is obtained via design experience or the given properties of the modeled IP.
3. Cycle approximate: the details of modeling are in the level of clock cycles. However, the number of clock cycles may differ from the actual implementation. Details such as pipeline stages when executing software are not considered. Clock cycles of software may be obtained by running each software process individually without the presence of operating systems.
4. Cycle accurate: very accurate instruction set simulator (ISS) or the actual RTL design of the processor model or hardware IP is used to execute software and to simulate hardware components.

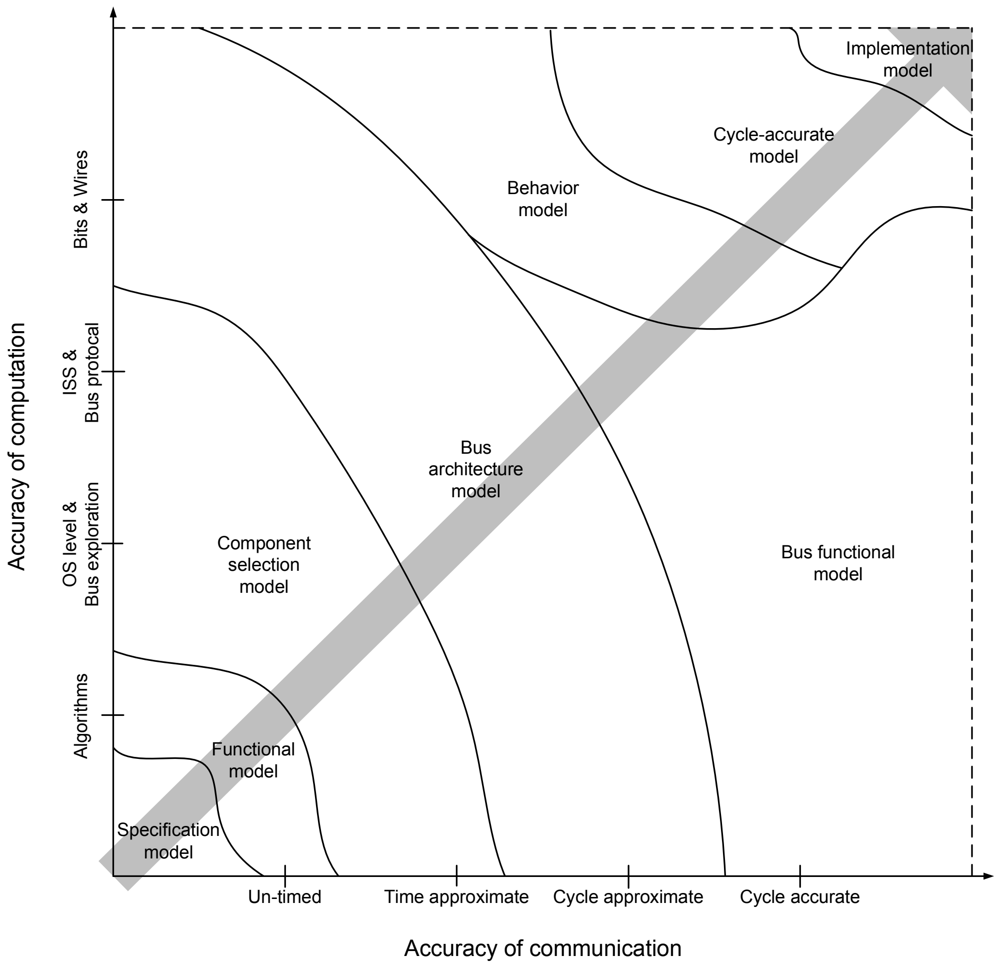


Figure 3.2: Modeling approaches at different accuracy levels

Data granularity in communication differs in how information is exchanged between components. The scenarios, from the most abstract to detailed are in the following:

1. Software to software: from unmanaged shared memory, to message-passing mechanism governed by the OS.
2. Hardware to hardware: from point to point channels, to packets transferred on a bus modeled without protocol but with an arbitrator, and to bit-true data transfer with dedicated bus protocol.
3. software to hardware: from modeled software that communicates directly with modeled hardware to using a device driver managed by OS to access hardware components from software processes.

Timing accuracy is co-related to data granularity in communication modeling. For instance, a functional bus model which operates according to a specified protocol synchronizing with a dedicated clock is modeled in a cycle-accurate manner.

Timing accuracy of computation is achieved in the same fashion as in communication. Data granularity used in modeling computation relies on co-simulation requirements and design refinements. The simulation speed benefits from the abstract computation model and is important in the early design phases. On the other hand, detailed data representation will be required for the implementation model. Hardware and software models exhibit different data granularity, from abstract to detailed, as follows:

1. Hardware: from functional description to RTL behavior model, and to cycle-accurate model or actual implementation
2. Software: from algorithm (or communicating behaviors, CB), to processes supported by OS (operating system level, OSL), to instruction level (IL), and to bit-streams of codes executed by the RTL processor models or real PEs (processor register transfer level, P-RTL)

Similarly, timing accuracy and data granularity in computation influence each other, as in communication models. Modeling approaches based on different levels of accuracy are illustrated and proposed in Figure 3.2, and called a ‘modeling graph’. Timing accuracy is used to represent abstractions of communication, as the horizontal axis of Figure 3.2. The vertical axis represents the data granularity at different degrees of accuracy.

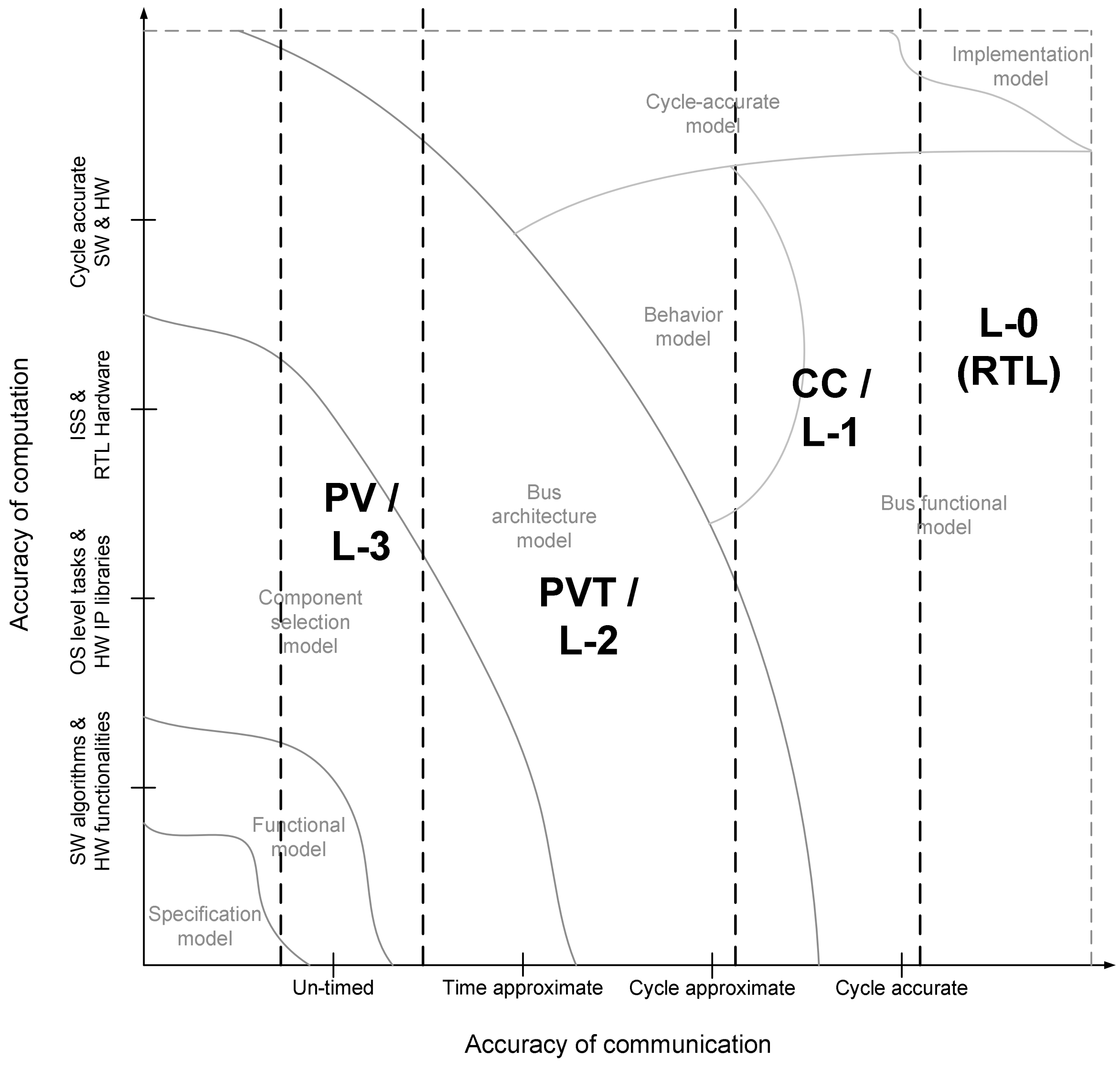


Figure 3.3: Coverage of transaction-level modeling

A single modeling technique is not sufficient to cover the whole design space. [Cai & Gajski, 2003] present models which are discretely distributed according to the levels of accuracy. Eight modeling approaches are illustrated in Figure 3.2: specification model, functional model, component-selection model, bus-architecture model, bus-functional model, behavior model, cycle-accurate model, and implementation model. The development path, shown as the grey arrow, originates from the specification model from the bottom left corner of Figure 3.2 and finishes in the implementation model at the top right of Figure 3.2. The path taken from the specification model to the implementation varies between different design approaches.

Transaction-level modeling (TLM) enables communication and computation to be modeled separately [Ghenassia, 2005]. TLM is used with other benefits such as: 1) early performance estimation in the timed model, and 2) higher simulation speed due to the higher level of abstraction. TLM provides a set of modeling approaches, which have been discussed in [Grötker et al., 2002], [Haverinen et al., 2002], [Connell, 2003], [Ghenassia, 2005], [Yoo & Jerraya, 2005], and [Black et al., 2008]. Different levels of abstractions in TLM are identified in different approaches as follows:

1. The modeling approaches are first grouped to ‘untimed’ and ‘timed’. Timed modeling is generally evolved from the untimed model by adding timing information. The untimed model includes a programmer view (PV), while timed models consist of a programmer view with timing (PVT), cycle callable (CC) [Connell, 2003].
2. Based on the communication layers [Haverinen et al., 2002], from abstract to detailed: message layer (L-3), transaction layer (L-2), transfer layer (L-1), and RTL layer (L-0).

The above approaches shared common features and are grouped and illustrated in Figure 3.3, which is based on Figure 3.2, presenting applicable TLM for staged design models. PV, PVT, CC, and RTL are names used to address the underlying models. TLM focuses on the communication between modeled components. Therefore the modeling graph is partitioned horizontally according to the characteristics of different TML approaches.

The system under design evolves from the specification model to implementation through communication and computation refinements. Descriptions of refinements in each model, along with the use of the TLM, are detailed in the following sections.

3.2.1 Specification model

The specification model is established with informal system descriptions. It does not contain any algorithms of the system under design, but consists of the requirements and constraints of the system. Examples of requirements are features such as what the inputs to the systems will be and how the system output is going to be displayed.

3.2.2 Functional model

The functional model is constructed from the specification model through the 'specification capture', as shown in Figure 3.1. The functional model is the executable version of the specification, in which behaviors (and possibly sub-behaviors within behaviors) of the system are identified. Behaviors are modeled as algorithms (an aspect of computation) which need not be detailed and used in the final implementation, but are sufficient to capture the corresponding activities of the behaviors. The functional model is usually single-threaded, in that concurrent behaviors are not yet identified. Behaviors and sub-behaviors are in the form of function calls. Communication between behaviors is via variables and argument-passing of function calls; hence the untimed nature of the model. The functional model is also called 'SoC functional view' [Ghenassia, 2005]. TLM-PV and/or L-3 are used in the functional model.

3.2.3 Component selection model

The component selection model is close to 'IP-assembly model' [Cai et al., 2003], 'component assembly model' [Ghenassia, 2005], and 'SoC architecture view' [Ghenassia, 2005]. Components in this model are mapped from behaviors identified from the functional model. Components can be existing software or hardware IPs, or IPs which will be designed manually or synthesized automatically in the later design stages.

Existing IPs can be either proprietary or from sources (or hardware descriptions in HDLs) available. Proprietary IP models may be at different levels of abstraction in both computations and communications. Therefore the component selection model covers computations modeled from functionality to detailed software library codes and RTL hardware, with both untimed and time-approximate communication. Since communications in the component selection model are point-to-point linkages, there is no presence of a bus in this model. Timing estimation can be annotated to mimic delays. TLM PV (L-3) and/or PVT (L-2) are used to describe communications. The component-selection model is the starting point of architecture exploration. [Séméria & Ghosh, 2000]

3.2.4 Bus architecture model

Further down from the component selection model, the bus architecture model presents a primitive description of a bus model hosting the interconnections between components. In this model, components which share information are coupled with the same bus. The architecture exploration is carried out to obtain optimal configurations between components-to-use and how connections between components are established by the means of a bus. In this model the bus will be refined to a hardware bus or a mechanism provided by OS for software processes, to communicate. Flexibility of this model is required to perform efficient architecture exploration; the protocols of buses are therefore absent. The bus architecture model is adopted by both system-level synthesis and component-based design to find the ‘golden model’, and the coverage of abstractions in both communication and computation is therefore vast. TLM PV, PVT and CC techniques are used in communications between components of all degrees of accuracy. The bus architecture model is similar to the ‘bus arbitration model’ [Cai & Gajski, 2003], and the ‘SoC architecture view’ in [Ghenassia, 2005].

3.2.5 Behavior model

The behavior model (BM) emphasizes the descriptions of components and the buses interfacing them. Descriptions are pin-accurate, and operations of components and buses are based on clock cycles. Since clock cycles in the behavior model are not as accurate as those in the cycle-accurate and implementation models, communications are modeled cycle-approximately. Clock cycles in BM are used mainly to activate state transitions of the components and the buses. Therefore the behavior model is suitable for refining the components and protocols of the bus. Computations in this model can be behavior descriptions of hardware components in RTL, or different software pieces running in an instruction set simulator (ISS) without the presence of OS to evaluate the performance of each process. Cycle callable (CC, L-1) of TLM is used in this model.

3.2.6 Bus functional model

The bus functional model (*BFM*) is equipped with a cycle-accurate bus model with specific protocol. Bus functional model evolves from the bus architecture model where the golden model of platform has been obtained. In the BFM, components interact based on the given protocol. Computation refinements of components are performed at this stage. Components at different refinement iterations are at different levels of abstractions. Techniques of inserting adapters, wrappers, and converters between the components and the bus are used to bridge the differences such as timing accuracy. Communications of BFM focus on cycle-accurate descriptions as illustrated in Figure 3.2, where CC (L-1) and RTL (L-0) are used to model communications. Computations of BFM are across a wide spectrum as shown in Figure 3.3, representing the refinements of components.

3.2.7 Cycle-accurate model

The cycle-accurate (CA) model is a pin-accurate and timing-precise model. Hardware components are modeled in synthesizable RTL and software processes are executed on the prototyped platform such as the one crafted in FPGA or the pre-existing development board, or the processors modeled in the RTL. Communication between components can be cycle-approximate in communication, and thus modeled with the TLM CC (L-1), if computations are verified on platforms differing from the final implementation. For example, a dedicated bus is required to link the FPGA-based processor model to the RTL hardware simulator to perform co-verification. Similarly, a fully accurate communication and computation model using RTL (L-0) will be adopted when the system under design is close to the final implementation, presented with the implementation model in the next section.

3.2.8 Implementation model

The implementation model is the last stage towards the final product of the system under design. Both communication and computation are modeled in the most detailed manner. Software is executed on the finalized platform for verification. Designers generally do not work directly at this level because the SW and HW components are compiled and synthesized from higher-level description unless specified in the design.

3.3 Operating systems in system-level design

3.3.1 System modeling with operating systems

With ever-increasing use of software in computer systems, OSs play an important role in a large class of computer systems. Characterization, and modeling, of the OS before carrying out the actual software implementation are essential for system development. Modeling OS as a part of the overall system model thus becomes essential. In order to estimate system performance, system designers should be able to model an entire system with the existence of an OS model prior to system implementation.

Since performance and the ability to meet time constraints of process execution rely not only on the processor power but also on how processes are managed by the OS, well-performed design of the OS becomes important. Strategies and mechanisms provided by OS must be taken into consideration when designing (or choosing) an OS. Introducing specific services to the OS can provide more efficient application development and can lead to more rational and 'build-by-correctness' designs.

Operating systems are introduced to system modeling because:

1. They provide interfaces for software processes communication and synchronization.
2. Access to hardware is provided as device drivers built-in or constructed on top of the OS.

3. Behaviors of software processes, such as execution order, are heavily influenced by the scheduling policy provided by the OS.

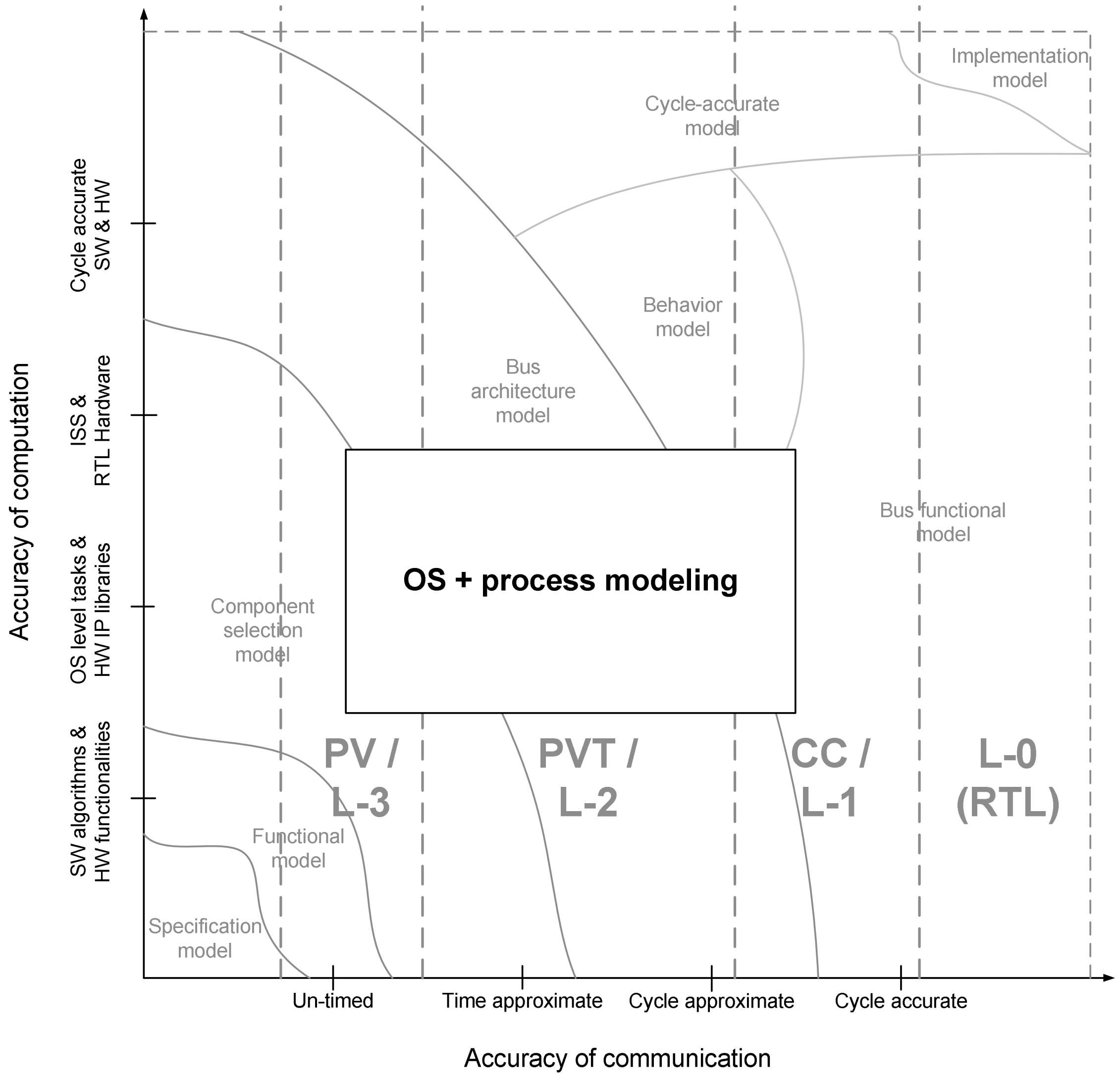


Figure 3.4: OS and software processes modeling in system design

Introducing OS to system design means having software processes implemented by using services provided by the OS. However, the process/thread-based software model does suffer for a variety of reasons, e.g. the race condition which is an obvious example of indeterminism introduced by software [Lee, 2006]. System models consisting of OS and process models thus inherit the same problems. Synchronous languages are proposed to resolve such problems, but have poor interactions with other software components and do not integrate well with the existing SLDL-based HW/SW co-design environment.

There are approaches which model software concurrency without having OS in mind but make use of the underlying simulation kernel. A modeling strategy that maps concurrent behaviors to processes/tasks is demonstrated in [Tomiyama et al., 2001]. The processes/tasks are assigned with fixed priorities, higher priority processes pre-empting those of lower priority. Pre-emptions are achieved by placing processes functions in a specific order in the system description. This approach limits the supported scheduling policy and the scalability of the model. Omitting the OS model in the software simulation does not provide sufficient information for OS mapping in the later design stage.

Simulation is one of the features benefitting from the OS model. It is possible to map the OS models to the existing OSs, or to generate customized OSs according to the application requirements. For example, automatic generation of RTOS proposed in [Gauthier et al., 2002] provides options such as processes communication, synchronization, and hardware requests to construct a customized RTOS. Such anapproach prevents unnecessary effort for RTOS porting between different applications and hardware architectures.

OS models, which are used in different design stages, are described in different levels of abstractions according to the modeling requirements of the development phases, as shown in Figure 3.4. As mentioned before, OS and software processes as components of a system, are well fitted into the staged system design.

OS and software processes can be modeled in a functional manner, and are refined in the final implementation. An OS model can be un-timed, for example, providing functionality to schedule processes to obtain software concurrency, regardless of the timing requirement. It can also be modeled in a timed fashion, where scheduling policies to achieve ‘real-timeness’ are modeled. In terms of modeling accuracy, an OS can be described in source code form of the SLDL, or the actual OS source can be used if available, depending on the simulation requirement. If the OS is available only in binary form (e.g. binary library for the simulation host or target platform), the OS library is linked with other software components, and is simulated as a host process or within the instruction set simulator (ISS).

When OS modeling is adopted in system design, various combinations of interactions between components exist. Software portions of the system are often divided into processes and OS, and such simulations have been carried out, such as processes/tasks modeling in [Poplavko et al., 2003], along with RTOS modeling in [Madsen et al., 2004] and [Gerstlauer et al., 2003]. Interactions between software programs and hardware devices are presented in [Honda & Takada, 2003] and [Formaggio et al., 2004].

3.3.2 Existing approaches of OS modeling

Techniques of modeling general OS, embedded OS, and RTOS have been discussed in [Yoo et al., 2002] as follows:

1. Mapping processes of the target platform to processes of the simulation host, also known as native simulation. For example, [Bouchhima et al., 2004] focus on the HW/SW co-simulations on arbitrary levels of abstraction, where interfaces/adapters are dedicated between software and hardware components in the model. µVirtualChoices presented in [Tan et al., 1995] is a simulation environment for the kernel µChoices on the Unix-based hosts. Emulating interrupts as UNIX signals is one of the approaches which maps hardware-dependent OS codes to resources available on the host OS. The rest of the OS codes are linked with the host OS counterparts to perform native simulation. The software processes are mapped to user-level threads of the host kernel where simulation is carried out.
2. Compiling the target software sources with the OS to an executable of the host platform. In [Yoo et al., 2002], the (RT)OS simulation model is generated from the actual (RT)OS. Software processes are modeled using threads of host OS and they communicate with each other through remote procedure calls (RPC). Hardware is also described in SYSTEMC. Estimation of software execution times, on the targeted (RT)OS, are annotated to the simulation model. This model evolves in [Yoo et al., 2003] so that time-delay functions are used to

synchronize both software and hardware simulation. Simulation of software works either with OS codes or scheduling mechanism provided by SYSTEMC. This approach requires the actual (RT)OS that is not suitable for design space exploration (DSE) where the (RT)OS selection may not be final.

3. Executing software processes with support of the virtual OSs. A virtual OS is the functional and abstract model of the real OS, and is intended to validate and simulate with the other software and hardware components. Virtual OS needs to provide the following:
    a. Interfaces to access features provided by the (RT)OSs. Interfaces can be in the form of function calls, signals, and events to the (RT)OS model. Interfaces remain while the underlying (RT)OS model can be inter-changed with other (RT)OS models, both at simulation and implementation.
    b. Essential features of (RT)OS, or acting as an intermediate layer to the existing (RT)OS codes.

As an example, [Zabel et al., 2009] present an abstract RTOS library, called aRTOS, which provides a set of interfaces to model processes and interrupt service routines (ISRs) using SC_THREAD of SYSTEMC. The designer can replace internals such as scheduling policy to mimic behaviors of different (RT)OS. In [Tan et al., 1995], the object-oriented µChoices is modeled as a set of objects which interact with each other, where a lower level nano-kernel is mapped to a host process. [Desmet et al., 2000] present SoCOS, a C++ based simulation environment that facilitates functionalities of (RT)OS. On top of SoCOS, OsAPI provides a generic interface for software to access the (RT)OS functions. OsAPI remains in the final implementation where SoCOS is replaced by the actual (RT)OS. Virtual OS simulation is to achieve performance speed-up comparable to simulating software in ISS. It also enables (RT) OSs to be modeled at different levels of abstraction. How processes are modeled depends on the features provided by the (RT)OS model. [Desmet et al., 2000], [Yoo et al., 2002], [Zabel et al., 2009] fall into this category.

Processes are mapped to constructs provided by the modeling language. For example, SC_THREAD from SYSTEMC is used to model processes in [Le Moigne et al., 2004]. A hardware/software modeling framework in [Chevalier et al., 2006] provides ‘swappable’ software and hardware partitioning at the early design stage by using a layer between the user module and the RTOS to simulate software and hardware interactions. However, the real RTOS has to be ported to a particular (target or simulation) platform. [Madsen et al., 2004] provides extensions made to SYSTEMC primitives to describe RTOS behaviors.

In [Le Moigne et al., 2004] the RTOS is modeled along with processes using two approaches: (1) both the RTOS and processes are modeled as threads in SYSTEMC and (2) the RTOS is described as a set of functions which will be used in actual programs of processes. In [Gerstlauer et al., 2003], interface of the RTOS model focuses on process creation and management, event handling, and time modeling. A process has to explicitly declare behaviors like fork and join through the process management interface. Process synchronization is implemented via channels between processes. Efforts are required to create dedicated channels whose number may eventually become very large and hard to organize. Moreover, resource sharing, an important feature of the RTOS, is not clearly presented.

Code generation for (RT)OS is also developed. [Gauthier et al., 2002] presents a methodology to generate an application-specific OS based on the requirements of the underlying application. Services provided by (RT)OS are differentiated and are stored in the form of source-code libraries. Existing (RT)OS requires effort to be merged into the library. Porting is still necessary for different target architectures.

3.3.3 Modeling OS with hardware involvement and support

SLDLs provide the means of hardware/software co-design and co-simulation in various levels of abstraction. Approaches to interface hardware and software, and refinements of interfaces towards the final implementations, are also proposed.

[Bouchhima et al., 2004] focus on the HW/SW co-simulations on arbitrary levels of abstraction, where interfaces/adapters are dedicated between software and hardware components in the model. The software processes are mapped to user-level threads of the host kernel where simulation is carried out.

Various approaches to implement part(s) of the RTOS in hardware or with hardware support are proposed. Most of them are in the form of add-ons to the platform processor(s). A high-performance communication manager in hardware cooperates with the on-chip processor in [Shalan & Mooney III, 2002]. [Lee et al., 2003] proposes a mechanism to synchronize critical sections of the executed code. A RTM (Real-time Task Manager) proposed in [Kohout et al., 2004] is an example of co-processor hardware support to achieve more efficient process scheduling. [Nakano et al., 2002] describes STRON-I, a design flow to migrate event flags, semaphores, timer, scheduler, and the interrupt mechanisms, to hardware. As an extreme, [Adomat et al., 2002] proposes RTU as an external hardware dedicated to perform RTOS functions in hardware. However these approaches are based on existing RTOSs or platform architectures, where modeling these components in higher levels of abstraction, which is important in the early design phase, is not presented.

3.4 The proposed system model with OS modeling

SystemC is used as the backbone of the proposed modeling framework. The main reasons for using SystemC are (1) it allows the modeling of system components regardless of hardware or software implementation and (2) it allows mixing with components developed in other specification languages, particularly HDLs.

The current version of SystemC lacks support for OS features. Besides that, using an existing OS implementation (where porting is required when running simulation on a host) in the early design phase would be overly complex and result in longer simulation time because of the execution of the OS code. Also, this approach is not flexible since some features, such as context switching, used by the OS are always platform dependent.

The OS modeling technique should enable both software (processes and the OS) and hardware components to be described and simulated together. The methodology introduced in this chapter also provides exploration of features of the OS. The essential features of this model should support the following:

1. Concurrent software processes are modeled in different SYSTEMC modules. Having all processes modeling in the same SYSTEMC module requires all of the relevant sub-functions to become member functions of the same module. This leads to poorly organized module description.
2. The OS needs to be modeled as a process in a dedicated module. OS can be seen as a program whose execution masters the overall software execution on its resident processor. Therefore OS can act as the bridge to the processor and other hardware components. Parts of the OS, such as services provided, are executed concurrently, if allowed by the platform. Furthermore, interrupts, which are handled first by the OS, can be modeled as the input to the OS which will trigger actions performed by the OS.
3. Hence, dedicated ports/interfaces of the OS module must be provided, to allow communication and synchronization between the OS module and other system components, e.g. programming interfaces for software processes and signals for hardware.
4. Internally to the software running on the same processor, generic interfaces of OS must be provided to elaborate process modeling/implementation with the least dependency on a specific OS.
5. The OS module should be a composition of sub-modules modeled as services or extensions based on the core-functionality (as another sub-module) provided by the OS.
6. The internal behavior of the OS, such as the scheduling policy, can be changed with no substantial effort to provide OS exploration in system design.
7. This model should be generic and thus able to be further mapped on standard/customized OSs whose performance can be evaluated.

The OS and software processes which are modeled as different design entities (SYSTEMC modules) are detailed in this chapter. The OS model receives events via different signals and reacts differently to each of them, indicating that the signals are the key object for the proposed OS modeling at the higher level of abstraction. Contexts of processes are stored within the body of the software process module, which enables shorer simulation times than simulating the actual context switch at a lower level. Processes are dispatched by notification from the OS model. Details of the software model will be described in later sections.

3.5 Service-based OS modeling with reactivity

In this section, an OS model which provides a set of services is presented, shown in Figure 3.5. OS services are accessed through application programming interface (API) by software. Proposed 'signal-operation services', which carry out operations on signals, are used to model synchronous models of computation. Also, the modular OS model allows substitution/support of its functionalities by specialized hardware.

The OS model consists of the following components, as illustrated in Figure 3.5:

1. Interface of the OS to communicate with software processes or the external environment, in the form of API (to handle requests from processes) or signal handler (for external signals)
2. A set of services with their own data structures. The current OS model has a set of core services (in the rounded rectangle) which consists of four main services (in rectangles). Data structures, shown in dashed rectangles, are used and managed by corresponding services.

Services provided by the OS utilize corresponding data structures, which collectively represent the current state of the OS. OS data structures consist of a number of queues, condition flags, tables, and counters. OS services are divided into four categories:

1. Resource management, which models mechanisms like semaphore used to lock and protect shared resources.

2. Timing control, to perform functionality such as the pausing of a process for a specified time.
3. Signal operation, to support reactivity.
4. Process scheduling, which works as a core service closely related to the other services. Features such as process creation belong to this category.

Groups of the OS services are formed hierarchically, and are able to perform independently as sub-modules within the OS.

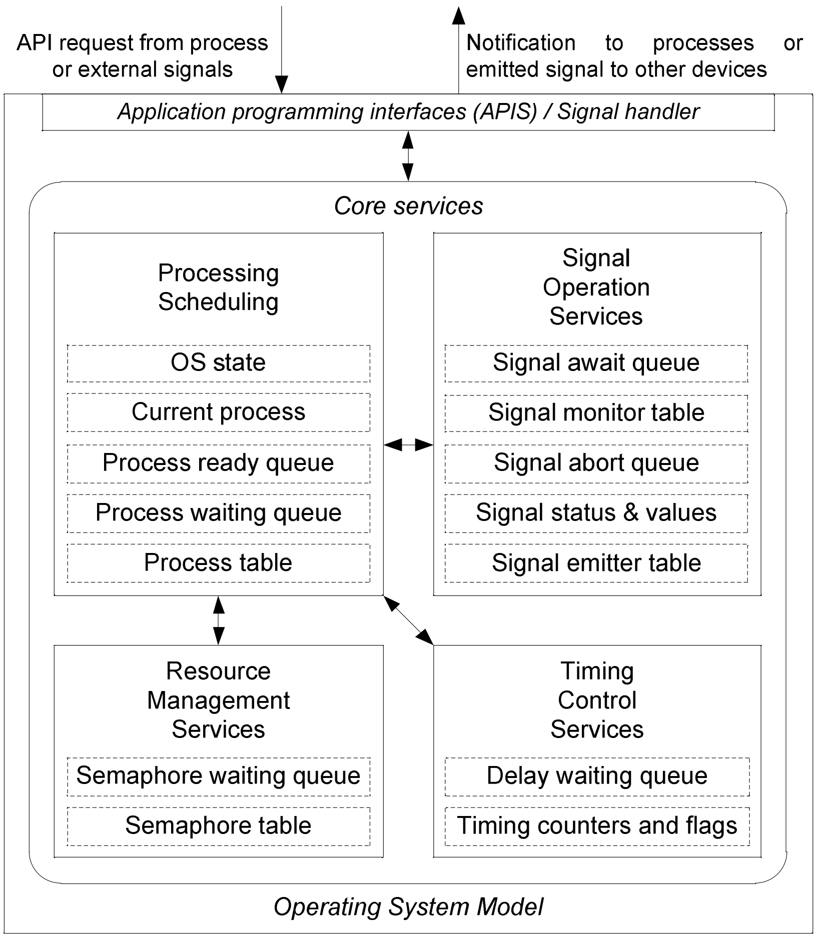


Figure 3.5: OS model including services provided and data structures

As an illustration, examples of the OS services for signal operations required in reactive systems are given in Table 3.1:

Table 3.1: API to perform signal operation

API	Descriptions
Signal_Await_Reg	Wait for presence of a designated signal.
Signal_Emit	Emit a signal to the other process or external environment.
Signal_Abort_Reg	Monitor a signal and jump to a specified address when the corresponding signal is present (implements pre-emption).
Signal_Monitor_Reg	Wait on the change of monitored signal.
Signal_Present	Check the status of a signal.
Signal_Value	Obtain the value of a signal.

Each group of services is supported by dedicated data structures. For example, signal-operation services are supported by data structures listed in Table 3.2:

Table 3.2: Data structure used by the signal-operation services

Data structures	Descriptions
Signal await queue	A list of suspend processes due to awaiting a signal.
Signal monitor table	A look-up table for monitored signals.
Signal abort queue	A list of pre-empted processes.
Signal status and values	The current status (absent or present) and value of a signal.
Signal emitter table	A list of processes associate with potentially emitting signals

Two methodologies of modeling the interconnections between the OS and other software components are presented in this chapter. One is pin-accurate modeling, the other is the transaction-level modeling (TLM).

Programmers obtain detailed views of software interactions in the pin accurate approach, where SYSTEMC primitive input and output signals are used. From the OS point of view, an incoming API call consists of the API type and arguments that are treated as input signals, whereas processes notification signals, which consist of process-ID and process-new-status, are considered as outputs, as shown in Listing 3.1.

Listing 3.1: Interfaces of process model at pin accurate level

```
#define ProcNum 16
#define API_Word_Width 16
#define API_Args_Width 16
#define Process_State 2
SC_MODULE(OSModel) {
  // Interfaces for accessing services – from processs
  sc_in<sc_lv<API_Word_Width> > process_API[ProcNum];
  sc_in<sc_lv<API_Word_Width> > process_API_Argument[ProcNum];
```

```
    sc_in<sc_logic> process_API_Last_Argument[ProcNum];
    // Interfaces for notifying processes
    sc_out<sc_lv<sc_logic> > process_Notification[ProcNum];
    sc_out<sc_lv<Process_State> > process_New_Status[ProcNum];
    // Rest of the OS model
    ......
}
```

API provided by the OS to processes can be divided into two groups: blocking and non-blocking. Processes will give the control to the OS or obtain control from the OS once the API calls are issued. Thus the APIs calls, as function invocations, are the linkage between the OS and processes. This is similar to transaction-level modeling (TLM). Both blocking and non-blocking interfaces are available in TLM and can be used to model API to access features of OSs. In this approach, the OS will be modeled as a SYSTEMC channel (a specialized module) which implements the interface (API provided) as services. Some of the OSs offer features such as modularity to include the essential mechanisms and services. Such OSs are generally modeled as hierarchical channels, the provided services being modeled as sub-modules within the channel representing the OS.

An API call with more than one argument in a pin-accurate approach consumes many clock cycles during simulation resulting in lower simulation speed. This is countered by introducing TLM, where the steps of passing API type and arguments to the OS are encapsulated within a single transaction. Interfaces of the OS model are bi-directional blocking interfaces, implemented in the OS as a SYSTEMC channel. Data types (classes) are created for service requests and OS responses, shown in Listing 3.2 with interface declaration.

Listing 3.2: Interfaces of the OS model in TLM

```
class OSAPI_if : public virtual sc_interface {
public:
  virtual NOTIFY service_request(const REQ&) = 0;
};
class REQ {
private:
  unsigned int API_TYPE;
  unsigned int *API_Arguments;
  unsigned int API_Arguments_Num;
  ......
};
```

```
class NOTIFY {
private:
  unsigned int Process_ID;
  unsigned int Process_New_Status;
  ......
};
```

Services are modularized according to the categories to which they belong. To achieve higher simulation speed, communications of grouped services are also described with TLM as shown in Figure 3.6.

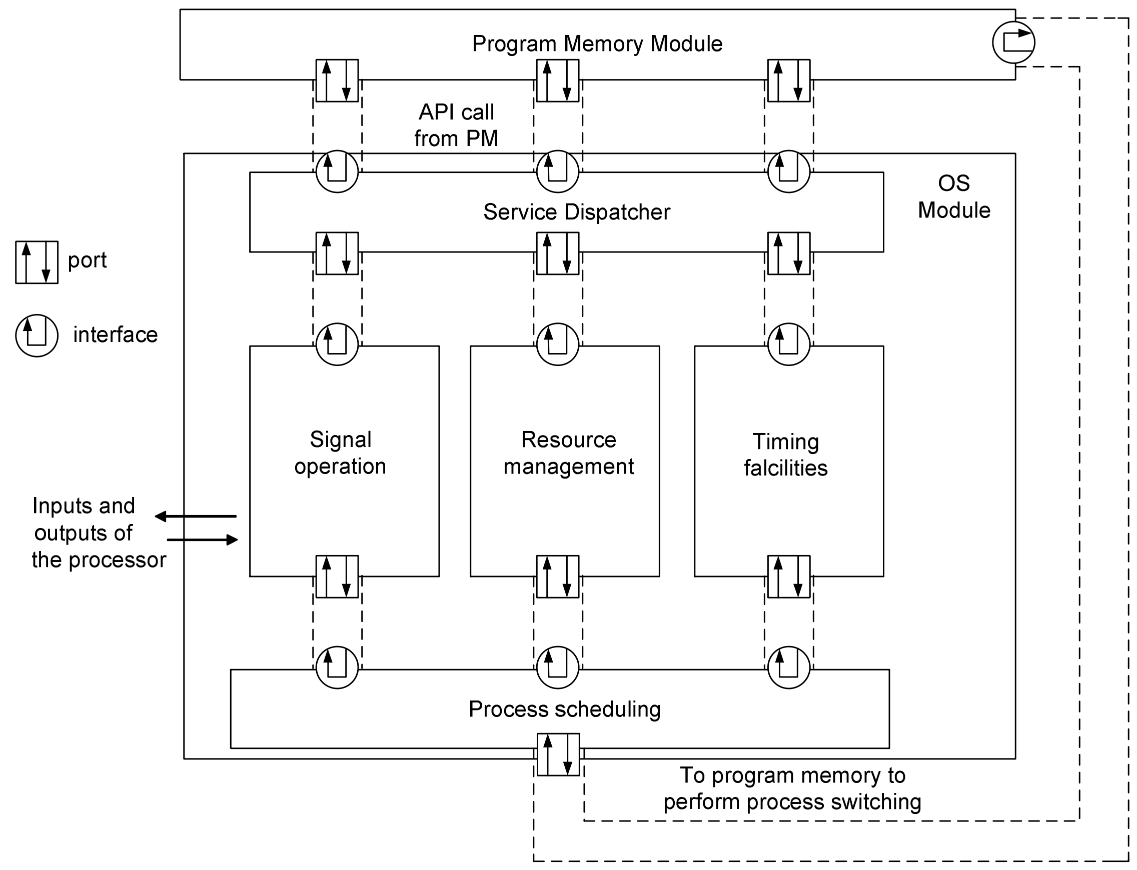
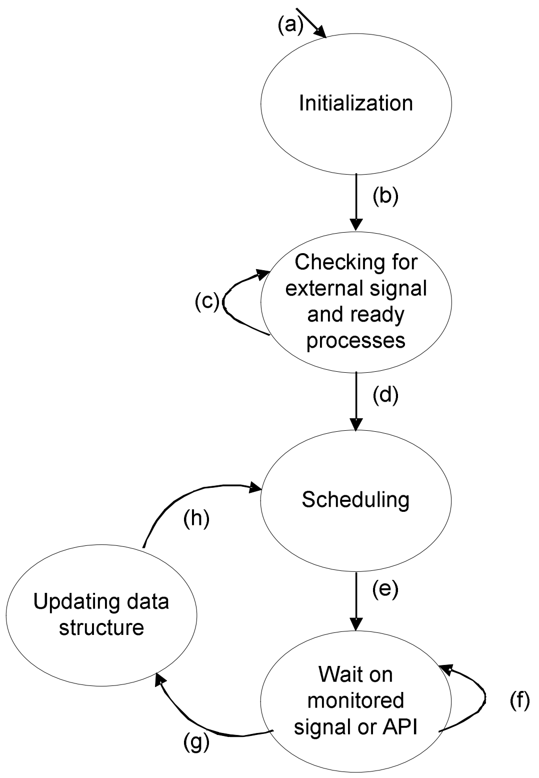


Figure 3.6: OS model in TLM

The OS model reacts to its inputs (service requests from processes) according to its state. The OS state transitions are illustrated in Figure 3.7 and Table 3.3. The OS state is part of the data structures governed by the ‘process scheduling services’, which coordinate the overall behaviors of the processes.

Table 3.3: State descriptions of the OS model

State	**Description**
a	Power up of system
b	Completion of the OS initialization
c	No ready process to release and no process activation signal is presented
d	Presence of a process activation signal or a ready process
e	A process is released
f	Neither an event nor an API call is detected
g	Receive an API call from a process, or a monitored signal presents
h	Finish updating data structure as preparation for scheduling


(a)
Initialization
(b)
(c)
Checking for external signal and ready processes
(d)
Scheduling
(h)
Updating data structure
(e)
Wait on monitored signal or API
(f)
(g)
Figure 3.7: State transitions of the OS model

3.6 Describing software processes with the OS model

3.6.1 Mechanism available in SYSTEMC to model processes

SYSTEMC provides processes (to differentiate from software processes in view of OS, it is called ‘SYSTEMC process’ in this thesis), namely SC_METHOD and SC_THREAD, which are scheduled in a co-operative manner. In other words, neither

SC_METHOD nor SC_THREAD are pre-emptive. SC_METHOD is suitable in modeling behaviors of a finite state machine (FSM), where each execution of SC_METHOD represents the activities performed in a specific/current state. On the other hand, SC_THREAD, is suitable for modeling software processes with numbers of segments formed by using wait statements provided in SYSTEMC. The wait statements are used to return the control back to the SYSTEMC simulation kernel manually so that SC_THREADs are scheduled co-operatively. SC_METHOD is used to model one software process, so that the process model can inherit the properties from the specification which can be described with formal MoC such as FSM. This approach also enables possible verification and linkage between other FSM based formal languages (such as ESTEREL).

3.6.2 Internals of the process module

A process modeled as a SYSTEMC module is called a 'process module'. It is specified with the process interface (to connect with the OS module), behavior (an SC_METHOD which contains the algorithms that describe reactions to various input events) of a process, a process state, process state transitions, context of the process, and an execution-control variable (called process-execution segment ID). The process module requests OS services by sending API calls and required arguments to the OS module via the communication channel which exists between the OS and each process. The communication channel is modeled as an un-timed TLM function call, for faster simulation, or timed (cycle- and pin-accurate) depending on the required accuracy. The OS notifies (through signaling) a process module to change its state. A process state indicates the current state of a process as illustrated in Figure 3.8 and whose state transitions are detailed in Table 3.4.

By encapsulating a process context within the process module as member variables of the process module, minimal or no effort is needed to model the context switching. The process module includes the declaration of the module and the process body as shown in Listing 3.3 and illustrated in Figure 3.9. The process body is modeled by using SC_METHOD. A segment of the process body behaves according to the current state of the process module and the current position (point) of the execution (control) flow. This state-machine based approach enables (1) a structured description of the process behavior and (2) a straightforward mapping from state-oriented system specifications.

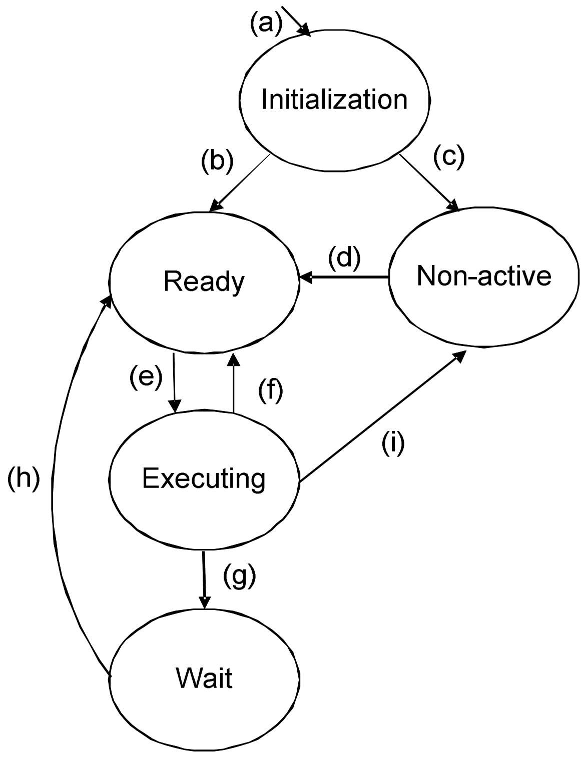


Figure 3.8: FSM of the process model

Table 3.4: State descriptions of the process model

State	Description
a	Power up of the system
b	End of process initialization, process starts immediately
c	End of process initialization, process waits to be activated by signal
d	Process activation signal is present
e	OS signals the process to be released or scheduled
f	OS signals the process to pause due to signal pre-emption or scheduling
g	An OS service is requested
h	Completion of a service call
i	Process termination

Listing 3.3: The SYSTEMC template of a process module

```
SC_Module(Process_Module)
{
  // Process module interface declarations
  Declarations of input ports;
  Declaration of output ports;
  Declarations of data structures including process context;
  Declarations of process simulation body function;
  SC_CTOR(Process_Module)
  {
    Assigning sensitivity list to the simulation body function;
    SC_METHOD(simulation_body);
    Initialization of data structures;
  }
}
void simulation_body()
{
  Process state transitions model
  // Process execution controls
  if ( process_execution_segment_ID = = segment1)
    Running process execution segment1;
  else if ( process_execution_egment ID = = segment2)
    Running process execution segment2;
    ......
    ......
  else if ( process execution_segmentID = = segmentN)
    Running process execution segmentN;
}
```

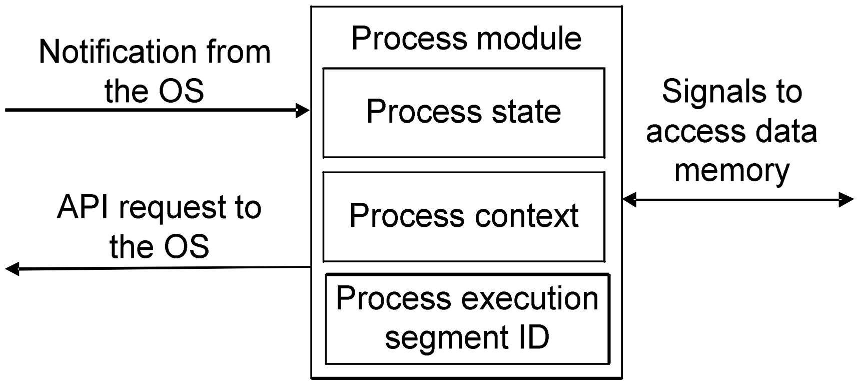


Figure 3.9: The internals and interface of a process module

3.7 Proposed co-design framework

3.7.1 The overview of the framework

The framework divides the design process into four stages as shown in Figure 3.10. At the first, a system is specified by using any available SLDL. System verification is then carried out to ensure correctness of the specification. The last step of the first stage is to analyze system behaviors and map behaviors to components. The HW/SW partitioning at this stage can be achieved by employing the designer's experience. There is no need for optimization since implementation details are not present in these HW or SW components (known as modules in the next design phase). During the second stage, embedded software is further refined into process modules and the OS module which is an abstract model without implementation details (such as disabling/enabling interrupts for critical sections, which are target-platform dependent). In this stage, exploration of HW/SW partitioning of the OS itself may be conducted. The hardware portion of the OS, along with the other hardware modules, including the processor module, memory modules, and other peripheral modules, is simulated (un-timed, or cycle accurate) with process modules and the software portion of the OS module.

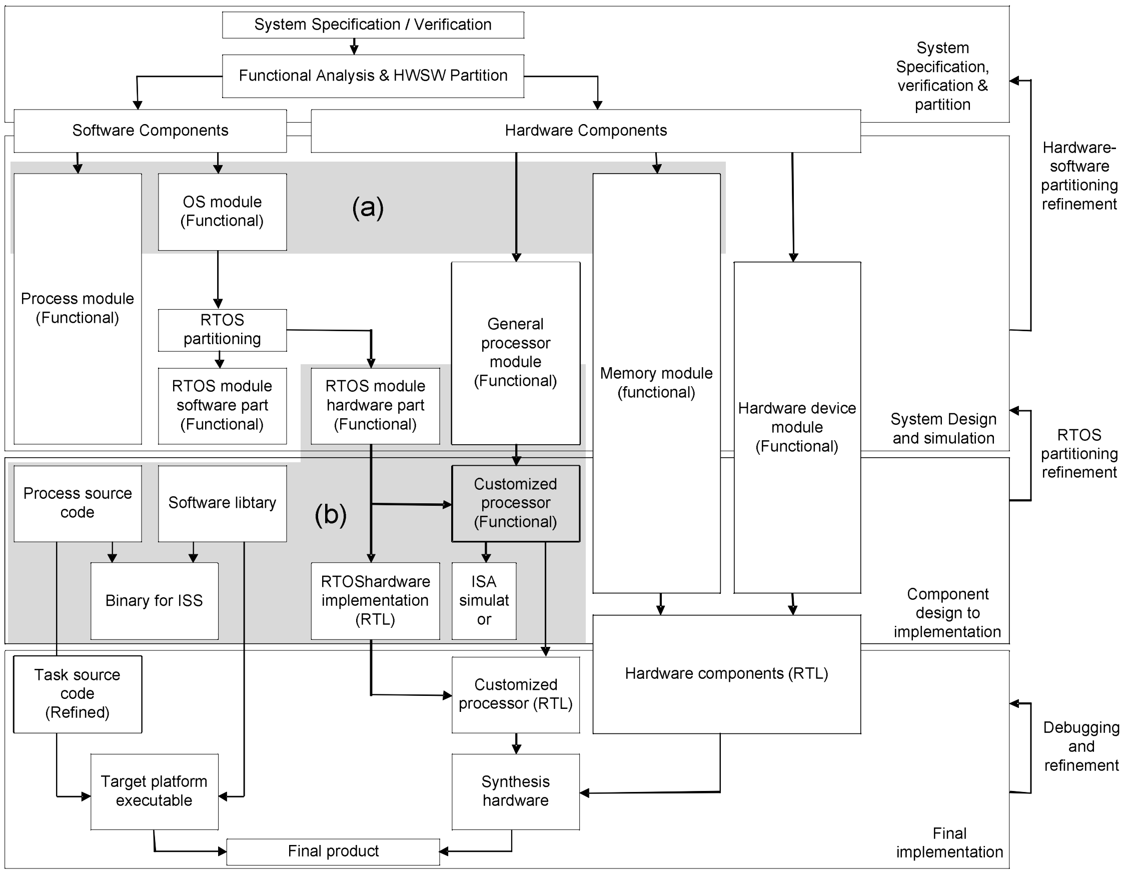

System Specification / Verification
Functional Analysis & HWSW Partition
System Specification, verification & partition
Software Components
Hardware Components
Hardware-software partitioning refinement
OS module (Functional)
(a)
Process module (Functional)
RTOS partitioning
General processor module (Functional)
Memory module (functional)
Hardware device module (Functional)
RTOS module software part (Functional)
RTOS module hardware part (Functional)
System Design and simulation
RTOS partitioning refinement
Process source code
Software libtary
(b)
Customized processor (Functional)
Binary for ISS
RTOShardware implementation (RTL)
ISA simulat or
Component design to implementation
Task source code (Refined)
Customized processor (RTL)
Hardware components (RTL)
Debugging and refinement
Target platform executable
Synthesis hardware
Final product
Final implementation

Figure 3.10: Modeling framework and staged design approach

Once the partitioning of embedded software and hardware devices is confirmed, the design process moves to the third stage, the implementation stage. Here implementation of processes and the selection of the OS are accomplished. The processor module is customized and integrated with the hardware portion of the OS.

Modeling of the dynamic behaviors such as the execution of concurrent processes is essential in the system model. Without adapting the OS in system models, different systems are modeled according to the underlying semantics of the used specification language. Simulators of the used languages will be required and will increase the complexity of the model. In contrast, if an OS model is used to control dynamic behaviors by providing semantic-preserving services, the system model is simplified.

Introducing the OS model into the framework can be seen as a bridge between the design and implementation phases, where communication and synchronization mechanisms are extracted from the behaviors in the specification stage and included in the OS model. In the rest of this chapter the focus is on the shaded area in Figure 3.10 (a) to explore the possible implementations with OS in the later stages. Process execution, which relies on support provided by the OS, is modeled as a process in SYSTEMC module. It is simulated together with the OS module. Information shared between processes is stored in the data memory described within another SYSTEMC module, which is also modeled in this stage.

3.7.2 Integration with the OS and process modules

This proposed OS and the process modules are integrated to represent the overall software components of the system. Co-simulations with hardware components and further refinements on components are in accordance with the aforementioned methodology. For example, programmers are interested primarily in communication between processes and their interaction with the OS, whereas system architects need a view from which to explore possible hardware/software alternatives. Figure 3.11 and Figure 3.12 illustrate two different approaches with the proposed OS and process models. During the early design phase, processes are modeled at the functional level (refer to Section 3.2.2), and executed with support provided by the OS through API calls, as shown in Figure 3.11. For example, API Signal_Monitor_Reg is used to monitor input signals. As an execution result, process switching would be required if a certain signal were present, where the scheduler takes the place of selecting the next process. Data memory, which is a functional model, provides temporary storage for process modules and the OS.

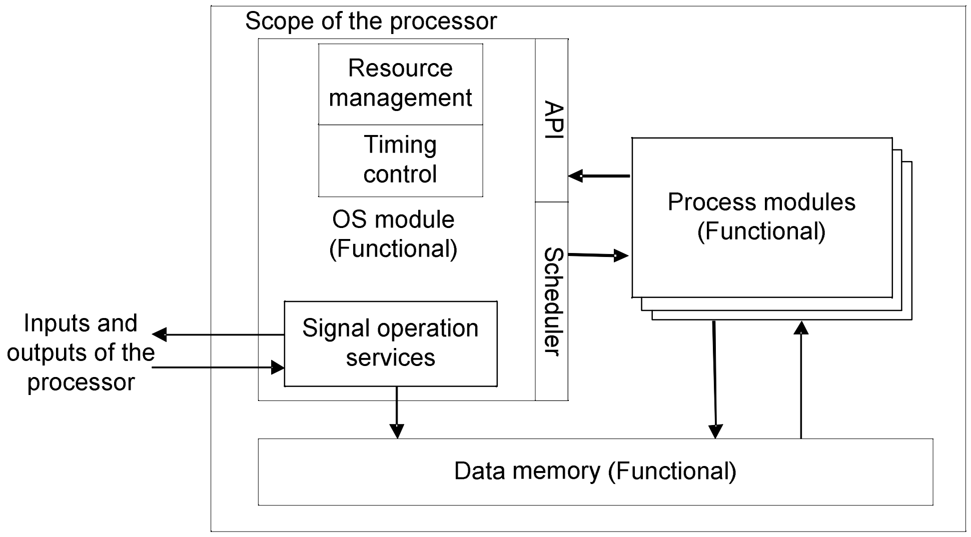


Figure 3.11: OS module with functional modeled processes

To explore OS with application-specific customization, an OS with different configurations (to provide different sets of services or to adopt different scheduling policies) is modeled. Simulation of pre-compiled processes is done by the ISS, as shown in Figure 3.12. This type of OS model enables designers to evaluate the OS design. Processes are first compiled with the skeleton of the OS library, which provides API only. Object codes of the compiled processes are stored in the functional model of the program memory, which fetches instructions to the ISS and the OS module. Before an instruction is loaded to the ISS, the program memory model checks whether the instruction is an API request. If this is the case, the request will be passed to the OS module instead of to the ISS. Once the request is carried out, the scheduler of the OS module notifies the program memory either to continue fetching instructions from the calling process (process continues to execute), or to load instructions from another location (as another process is released). Prior to the release of the scheduled process, a sequence of instructions is fetched to the ISS to simulate the context switching. The process contexts are stored in the data memory belonging to the processes. Data memory connects with the OS module, whereas device drivers, as a part of the OS, require memory access to control memory-mapped devices.

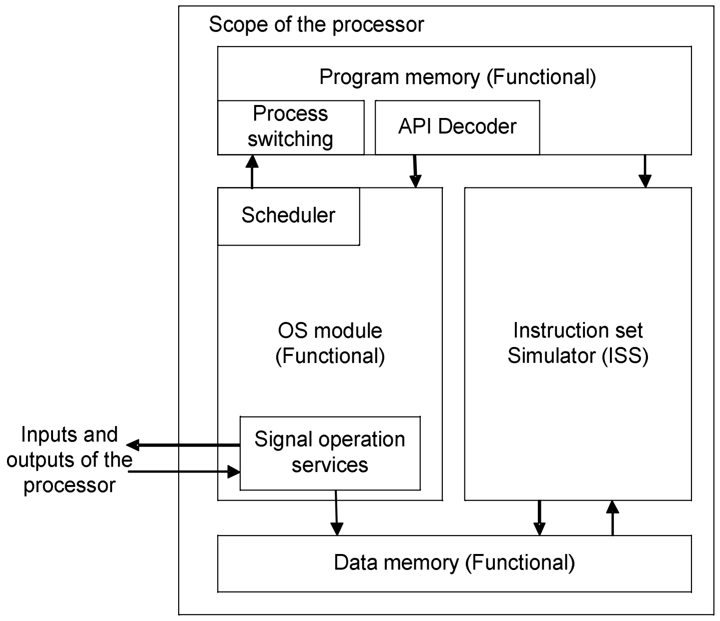

Scope of the processor
Program memory (Functional)
Process switching
API Decoder
Scheduler
OS module (Functional)
Instruction set Simulator (ISS)
Inputs and outputs of the processor
Signal operation services
Data memory (Functional)

Figure 3.12: The OS module with compiled program

3.7.3 Communication between modules and environment

Communication and synchronization of the process modules is managed through services provided by the OS. By adopting the use of the OS module, the number of communication links between the processes is reduced since the processes are communicating in a centralized fashion. Moreover, signals, which are used in many SLDLs and synchronous languages, are introduced as the basic communication mechanism to ease the transition between system specification and implementation. In this approach a process communicates with another process via internal signals (S_N) and with the external environment via external signal (ES_M) with OS support, as shown in Figure 3.13.

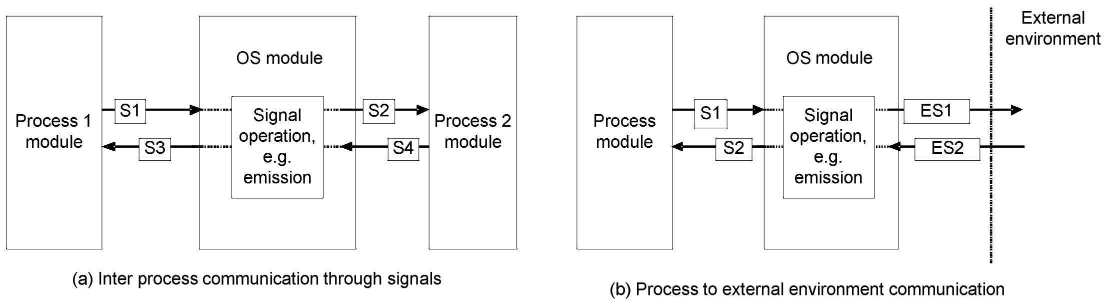

Process 1 module
S1
S3
OS module
Signal operation, e.g. emission
S2
S4
Process 2 module
(a) Inter process communication through signals
Process module
S1
S2
OS module
Signal operation, e.g. emission
ES1
ES2
External environment
(b) Process to external environment communication

Figure 3.13: Interactions between processes/external environments

3.7.4 HW/SW partitioning and HW support of OSs

A composition of processes and the OS is generally and commonly seen as software components of the embedded systems. In implementations, OS is presented in the form of libraries, which are used in compilation with process source codes to embedded software. Device drivers as parts of the OS libraries are provided to control devices, and thus the OS bridges the processes and hardware devices. It is also possible to implement partial OS functionalities in hardware to achieve higher performance, as described in Section 3.3.3 and this is why OS can be considered a mixture of hardware and software. To describe the behaviors within OS and explore HW/SW trade-offs in its implementation, the OS is modeled at the functional level with the aim of enabling co-simulations with other system components modeled in different levels of abstraction.

In the proposed system model, most of the components are described as modules in SYSTEMC, while others, for instance the processor, are either modeled in the register transfer level through hardware description languages (HDLs) or is presented through an instruction set simulator (ISS). Figure 3.14 illustrates how system components interconnect. The hardware/software composed OS connects to most of the other system components. Processor, memories, and devices communicate through a functional bus model. The processor model connects with the OS model, because the OS provides a platform-dependent layer such as drivers. Because OS functions implemented in hardware are integrated with the processor as functional units, the interconnections are presented. Program memory model connects to the $OS_{hardware}$ to provide information for executions of functional units. $OS_{software}$ processes service requests from compiled software processes, which are stored in the program memory. Data memory stores information which is manipulated by the OS and processes. Processes operate with support and services provided by the OS. Since data memory is modeled functionally, the details of timing in the memory model are not taken into account; the memory model can still be refined with further accuracy by back-annotating timing characteristics.

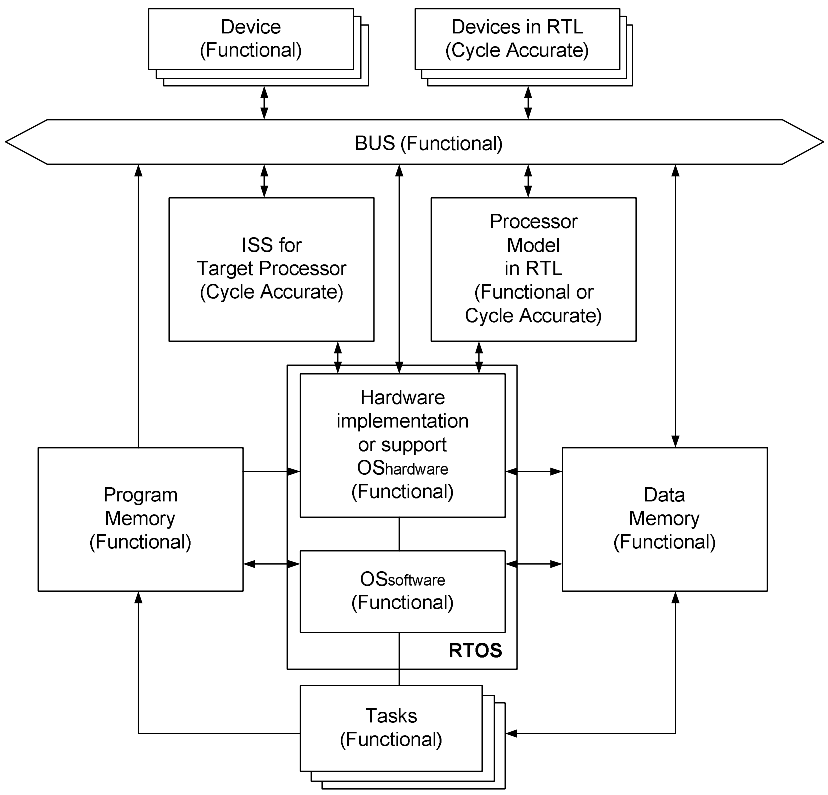


Figure 3.14: Hardware-supported OS in the system model with other HW components

The OS model is used to encapsulate the details of communication and synchronization between processes, whereas software developers need not to be concerned with how to maintain links between processes and other external devices. The OS model is close to the virtual OS simulation concept: a set of system services, available to processes through application programming interface (API) calls. Validation of the OS is achieved by examining interactions between the OS and other system components, and mapping applications on different OS configurations (with different services or HW/SW partitions) makes the approach independent of the target platform.

Data memory in this system model is currently modeled to store variables shared by the processes and values of memory-mapped signal values (a valued signal has status and value when it is present). Data memory is described as a SystemC module with interface to allow reading from and writing to an array whose size equals the addressable memory space of the target processor. A data-memory module connects with process modules in the system model. It is also used to provide information on process activities such as the use of shared resources.

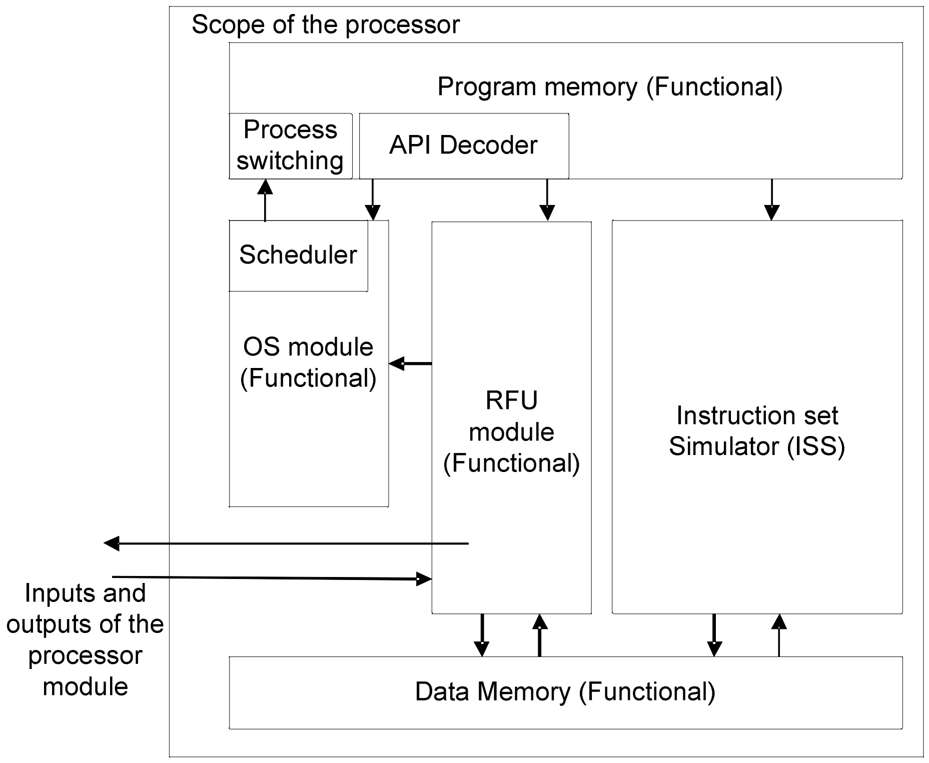


Figure 3.15: Processor model with RFU support

The proposed system model in this chapter allows (1) mapping of application specifications to software processes and (2) exploration of hardware/software (HW/SW) trade-offs in implementation of the OS. It also allows mapping on existing OSs and extending them with new signal-operation services. Based on the simulation results, possible and preferred configuration (HW/SW partitioning) of the OS implementation can be obtained. Hardware support to the OS can be integrated with the processor in the form of functional units. As an example, a Reactive processor [Salcic et al., 2005] contains RFU (reactive functional unit) to perform reactive operations on signals.

The OS model can thus provide us insight into migration services, in this example signal operations, from software to hardware, and model them as a hardware unit as shown in Figure 3.15. This model is derived from Figure 3.12 by introducing the RFU as a support to the OS. Instructions identified as signal operations, are fetched to the

RFU module. RFU is responsible for informing the OS model whether a process-scheduling is required at the end of a requested signal operation. RFU is connected with the data memory where process contexts relevant to the RFU can be saved/restored during the process switching.

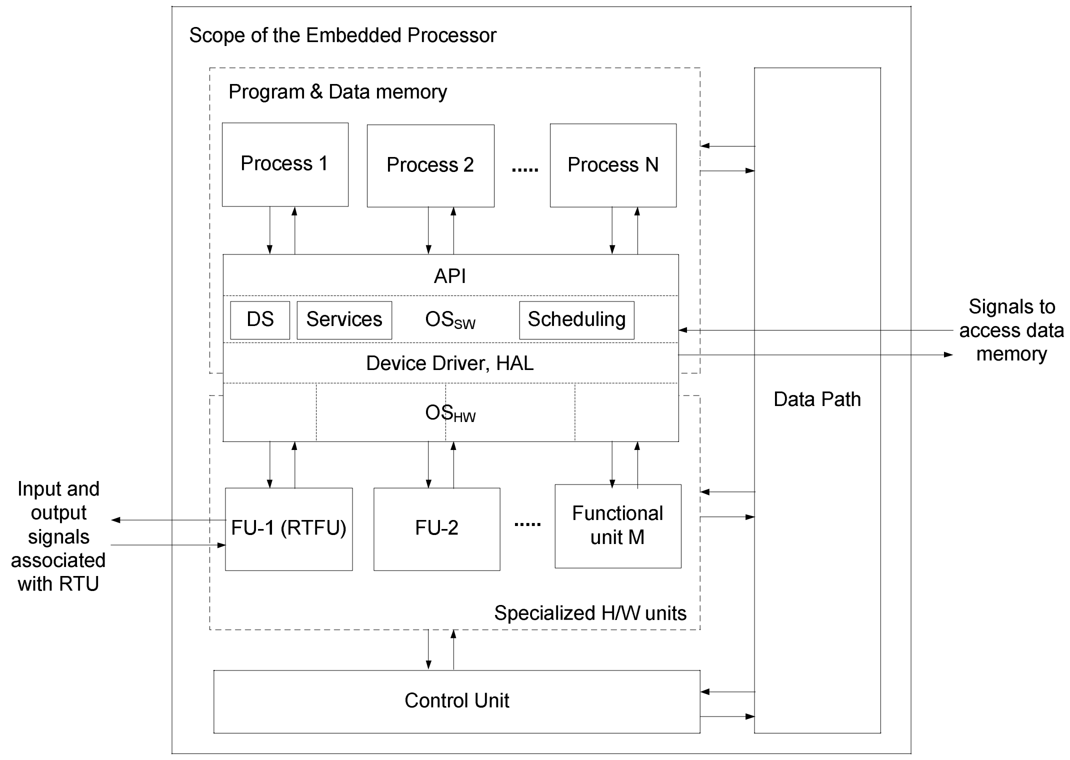


Figure 3.16: A model of processor: OS with hardware support

Figure 3.16 further details the processor model, which covers the overall software running on the processor and hardware enhancements to the processor itself. API calls to the OS are done by signaling, described in a later section. The API, OS_{SW} (which represents the software implemented part of the OS), device drivers, hardware abstraction layer (HAL), and processes are implemented in software which is compiled and stored in the program memory. Results of computation performed by the processes and the OS are stored in the data memory. The processes are allowed to perform operations on memory-mapped input/output signals via the support of OS.

3.7.5 Mapping of SW models to implementations

The proposed approach for modeling reactive embedded systems with an OS model can be easily extended to a real (target) OS by properly mapping signal operations on the services of the existing OS. For that purpose special signal-oriented data structures have been introduced. Each signal-operation service can be refined into two stages: (1) accessing and updating the semantics-related data structures (those preserving semantics of e.g. ESTEREL) and (2) calling other services provided by the target OS to explicitly trigger process-scheduling, if required. To fulfill these requirements, the target OS must provide the following services:

Entry and exit of a critical section – these are used to operate on the signal data structures, where operations should be carried out by one signal-operation service at a time. This requirement can be achieved also by using a binary semaphore which protects the data structure.

Blocking the process execution with time-out support – this can be a binary or counting semaphore, whereas the time-out feature is required to achieve exception/error handling such as a recovering from a signal non-presence within the specified time period.

As an example, two existing OSs have been used as the target OS: µC/OS-II and FreeRTOS. The templates of the signal-operation services for these two OSs are illustrated in Listing 3.4 and Listing 3.5, respectively. Note that the variable name timeOutLength is used to achieve the above-mentioned time-out.

In this model, the term ‘logical tick’ is adopted from the synchronous MoC, as the synchronization barrier of behaviors in the program execution. It is different from the standard OS ticks which are based on actual time (or clock cycles), and it is of different length in terms of execution time. Logical ticks are used to handle incoming events in a deterministic manner.

Listing 3.4: The template of signal-operation service for µC/OS-II

```
UserDefined_OS_Signal_Operation_Name (...)
{
  // Stage 1: Computation on data strucuture (DS)
  int blockingRequired = 0;
  // 0:non-blocking, 1:blocking due to signal, 2;blocking due to tick
  // obtaining the access of the DS
  OS_ENTER_CRITICAL();
  // Processing the DS, e.g. signal presence table, tick table, etc.
  blockingRequired = 0; // or 1 or 2 according to the computation results
  // release the lock to the DS
  OS_EXIT_CRITICAL();
  // Stage 2: Block the process execution if required
  if (blockingRequired == 1)
    OSSemPend( semaphore_for_signalS_of_procN, timeOutLength, err_code );
  else if (blockingRequired == 2)
    OSSemPend( semaphore_for_tick_of_procN, timeOutLength, err_code );
  // arriving here when the blocking is not required or finished
  blockingRequired = 0;
}
```

Listing 3.5: The template of signal-operation service for FreeRTOS

```
x_UserDefined_Signal_Operation_Name (...)
{
  // Stage 1: Computation on data strucuture (DS)
  int blockingRequired = 0;
  // 0: non-blocking, 1: blocking due to signal, 2:blocking due to tick
  // obtaining the access of the DS
  xSemaphoreTake( semaphore_for_data_structure, portMAX_DELAY );
  // Processing the DS, e.g. signal presence table, tick table, etc.
  blockingRequired = 0; // or 1 or 2 according to the compuation results
  // release the lock to the DS
  xSemaphoreGive( semaphore_for_data_structure );
  // Stage 2: Block the process execution if required
  if (blockingRequired == 1)
    xSemaphoreTake( sem_for_signalS_of_procN, ( portTickType )timeout );
  else if (blockingRequired == 2)
    xSemaphoreTake( sem_for_tick_of_procN, ( portTickType )timeout );
  // arriving here when the blocking is not required or finished
  blockingRequired = 0;
}
```

3.8 Case study: lift controller

A lift system from [Berry, 2004] was originally specified in ESTEREL. The system consists of a lift cabin, a set of sensors, a timer, three motors, few push buttons, a number of indicators (lamps), and a system controller. To map the specification to processes, primitive behaviors are first extracted from the system specification. Two large behaviors (call handling and cabin door activities) from the ESTEREL specification are decomposed into eight primitive behaviors and dependencies between behaviors identified as illustrated in Figure 3.17.

The primitive behaviors are mapped to 8 processes, where 17 existing dependencies (possibly require 17 communication channels if the OS is not used) are modeled with 8 communication channels connected to the OS module (un-timed). The processes are described using the timed model from Section 3.4. Example of mapping an ESTEREL description to the corresponding process segment is shown in Listing 3.6 and

Listing 3.7.

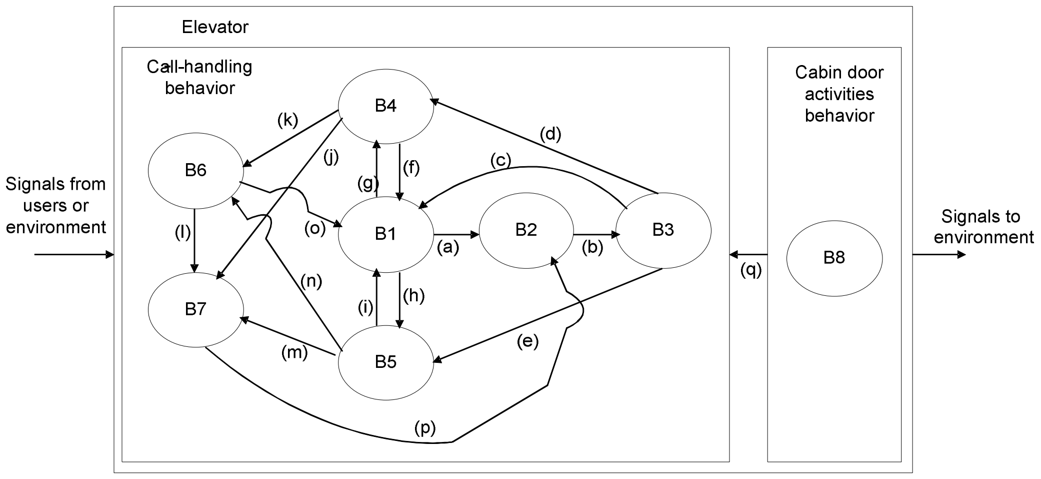


Figure 3.17: Primitive behaviors and dependencies extracted from the specification

Listing 3.6: Behavior described in ESTEREL

```
if (not StoppedAtFloor) then
  emit {
    PendingCabinCall <= CabinCall or ......
    PendingUpCall <= UpCall ......
    PendingDownCall <= DownCall ......
    PendingCall <= PendingCabnCall or
                   PendingUpCall or
                   PendingDownCall
  }
end if
```

Listing 3.7: Behavior description in SystemC

```
// Checking the presence of the following signals
int sStoppedAtFloor = Signal_Present(StoppedAtFloor);
int vCabinCall = Signal_Value(CabinCall);
int vUpCall = Signal_Value(UpCall);
int vDownCall = Signal_Value(DownCall);
int vPendingCabinCall = 0;
int vPendingUpCall = 0;
int vPendingDownCall = 0;
int vPendingCall = 0;
......
if(sStoppedAtFloor == 0) {
  vPendingCabinCall = vCabinCall ......;
  vPendingUpCall = vUpCall ......;
  vPendingDownCall = vDownCall ......;
  vPedingCall = vPendingCabinCall | vPendingUpCall |
                vPendingDownCall;
  Signal_Emit(PendingCabinCall, vPendingCabinCall ......);
  Signal_Emit(PendingUpCall, vPendingUpCall);
  Signal_Emit(PendingDownCall, vPendingDownCall);
  Signal_Emit(PendingCall, vPendingCall);
}
```

Figure 3.18 illustrates the model of the lift controller, where an additional process is introduced to function as the test-bench and simulates the external environment. The test-bench emits signals to the other processes through API calls, where emitted signals emulate inputs from the external environment. In order to observe the advantages of a modularized OS model, where services can be introduced and removed as the application requires, signal-operation services are removed from the OS model in order to analyze OS models with and without signal services. Two lift systems with the same functionalities were modeled and simulated. The first system model (Model A) achieves process communication and information broadcasting based on the use of semaphores. The second system model (Model B) is supported by signal operations provided within the OS.

Two models are simulated with 72 events (which may consist of more than one input occurrence) provided by [Berry, 2004]. Events are generated at random intervals. Simulation results are shown in Figure 3.19, where the bold line indicates the result generated from Model B, which has an average speed-up of 28.46 times in simulating clock cycles.

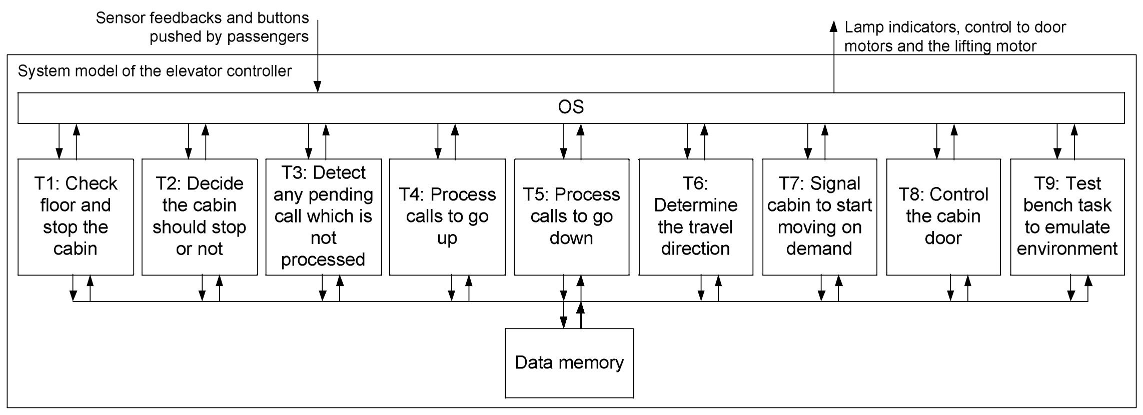

Sensor feedbacks and buttons pushed by passengers
Lamp indicators, control to door motors and the lifting motor
System model of the elevator controller
OS
T1: Check floor and stop the cabin
T2: Decide the cabin should stop or not
T3: Detect any pending call which is not processed
T4: Process calls to go up
T5: Process calls to go down
T6: Determine the travel direction
T7: Signal cabin to start moving on demand
T8: Control the cabin door
T9: Test bench task to emulate environment
Data memory

Figure 3.18: System model of the lift controller example

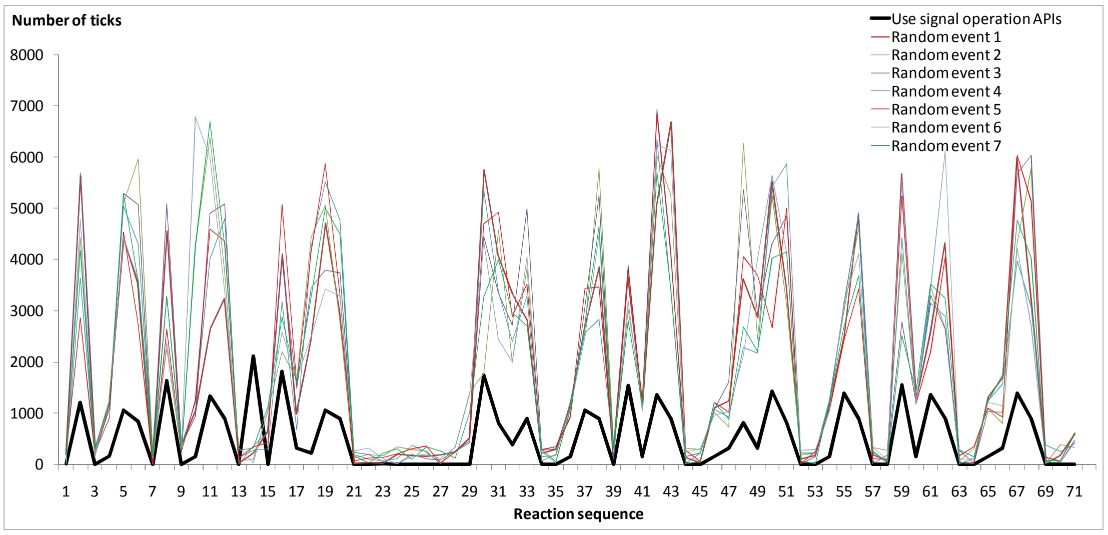


Figure 3.19: Simulation results for two system models

In Model A, synchronization occurs when a process releases a semaphore (use of a particular signal) where notification is sent to other processes. However, processes are notified regardless of the status or value of a signal. Checking of signal values happens each time the corresponding semaphore is obtained by the process. This creates a scenario where processes are polling signal values in a loop. In contrast to this, synchronizations occur through signal operations in Model B, where processes are notified when signal values change. The response times of Model A vary with event intervals, which are shown as multiple traces in Figure 3.19, whereas the response times of Model B are fixed in every simulation.

3.9 Summary

This chapter presents an approach to modeling the OS for reactive embedded systems. The abstraction level of the OS itself can be chosen depending on the need of the computer system designer. The OS model is modular and described in SYSTEMC, and gives opportunities for exploration of hardware/software trade-offs in implementations of the OS. The proposed signal operations in the form of OS services provide a mechanism to produce a relatively straightforward transformation of ESTEREL-like specification to processes, thus bridging the system specification and the design phase.

The processor model can be introduced into the system model in the form of an ISS or a low-level RTL model. Evaluation of performance or other aspects of the OS implementation in different configurations will be investigated. The future goal is to use the developed modeling methodology to explore OS customization for specific reactive-embedded applications.

The advantage of mapping dedicated services to support reactive systems has been presented in the case study. This was the stepping stone to the design and implementation of more powerful mechanisms for grouping of software processes in the form of a library, libGALS, which is built on top of the underlying OS and is presented in Chapter 4.

4

libGALS: a library for GALS system design

libGALS is a library and run-time environment that extends operating systems (OSs) to support the design of Globally Asynchronous Locally Synchronous (GALS) software systems and models. libGALS provides an application programming interface (API) that enables the designer to describe concurrent libGALS programs and reactivity in sequential programming languages. Moreover, it facilitates the interface between the GALS concurrent program and other processes through the services provided by the host OS. libGALS is also suitable as a target for code generation from GALS and synchronous concurrent languages. At the end of this chapter, experiments demonstrate code size and run-time gains when compared with other approaches to implement GALS systems.

4.1 Programming with a formal model of computation

The last decade has seen a huge growth in the complexity of software systems, which, due to the drawbacks of programming languages, usually do not follow any formal model of computation (MoC) and are therefore difficult not only to design but also to validate and verify. Programming languages like C/C++ and Java provide inadequate facilities to describe important behaviors of complex systems like concurrency, determinism and interaction with the environment. They require the use of operating system mechanisms that are available through the OS API, which require the designer to delve into low-level details instead of concentrating on the system design at hand. Since these mechanisms are not guided by any formal model, space it left to the making of erroneous designs. Synchronous languages like ESTEREL [Berry, 2000] and GALS system-level languages like SystemJ [Malik, 2010] have been shown to increase designer productivity when designing large and complex systems. They provide an abstract way to model concurrency and communication with the environment, besides being formally verifiable. However, they also have certain drawbacks such as:

1. Large generated code size.
2. Mapping of the concurrent programs onto single threads in the targeted OS environment. Current synchronous language compilers compile away the concurrency to produce a single-threaded C code. This generated code is unable to take advantage of the multicore processors, its large size and single-threaded nature slowing down the execution speed of the designed systems.
3. Lack of a formal communication model between the designed system and other parts of the system, which are asynchronous in nature (e.g. the device drivers which co-exist within the system).
4. Existing synchronous languages are too hardware-like for most software programmers. Interaction of a synchronous program with the environment is not addressed in a general way and the programmers must deal with it on case-by-case basis using low-level language abstractions.

In this chapter a software library and run-time environment for the execution of GALS systems, called libGALS, is presented. Concurrent behaviors are implemented as task-based software processes around an operating system, and they comply with the GALS MoC. libGALS provides a layer atop the host OS and/or a threading library such as pthread [IEEE, 2008]. It can be used to program concurrent systems following the formal GALS MoC as a safer alternative to conventional threading approaches. libGALS can also be used as a target for compilation of specifications in languages such as ESTEREL and SystemJ. Porting libGALS is easy and can be done for almost any existing OS. libGALS can then be used from sequential programming languages through a set of proposed API. The main novel features provided by libGALS, which also affect the way concurrent programs are written in sequential programming languages, are:

1. Ability to extend sequential programming languages such as C (through available language bindings) to specify synchronous type concurrency with simple mechanisms for communication and synchronization between synchronous processes using signals. Communication and synchronization between asynchronous processes are through channels implementing message-passing with rendezvous. Signals and channels can be created dynamically.
2. Ability to dynamically create processes and define their relationship with already existing processes (synchronous or asynchronous), as well as to dynamically schedule these processes. libGALS allows designers to create processes either dynamically or statically depending upon application requirements. In a safety-critical or sensitive application it would be prudent to create all processes at startup. The static creation of processes would allow the designed system to be analyzed for predictability and timing performance.
3. Provision of interface to the external environment through signal abstraction and to other OS processes through host OS services.
4. Achievement of higher responsiveness and reduced response times compared with current language approaches to GALS and synchronous concurrency.
5. Smaller memory footprint compared with other GALS approaches, thereby, making it also suitable for embedded systems.
6. Ability to define simulation model for modeling and simulation of complex system designs.

libGALS is entirely written in C and as such has great degree of portability to practically any host operating system. The first implementation presented in this chapter targets Linux, although there have already been ports to some other common OSs. Throughout this chapter, the term OS is used to represent common operating systems with sufficient features to support libGALS such as, but not limited to, Linux.

The rest of this chapter is organized as follows. Section 4.2 presents related work. Principles of operation and implementation of libGALS are given in Section 4.3. An example of GALS design is given to illustrate both specification and implementation features of libGALS. Section 4.4 presents performance comparisons with the GALS language SystemJ to indicate potentials of the proposed approach, not only as an alternative, but also as the way to merge those two approaches, using the GALS language on specification and the libGALS approach on the implementation level. Discussion and conclusions are given in Section 4.5.

4.2 Approaches in programming concurrency

4.2.1 Concurrent behaviors in software systems

Specification and run-time execution of concurrent processes are supported using different mechanisms. In an OS, concurrency is implemented in the form of multiple processes (sometimes called tasks) supported by a scheduler implementing switching between these processes to better use the processor and to provide faster response to the events from the environment. However, multiple processes require mechanisms for synchronization, communication and mutual exclusion for the protection of shared resources. OS [Silberschatz & Galvin, 1998] provides this support in the form of traditional API to programming languages. These mechanisms must be used by system programmers with due care to prevent non-deterministic or non-desired behavior and traditional pitfalls such as deadlock or race conditions [Silberschatz & Galvin, 1998]. Java provides native multithreading support, but the programmer is responsible for correctness of the program as it does not follow any formal MoC. Also, its concurrency is non-deterministic. Recently, OSs have been extended to support execution and concurrency in symmetric and asymmetric multiprocessor systems.

4.2.2 Limitation of single-threaded specification models

In system-level languages concurrency is described and dealt with using language features. When compiled, concurrent behaviors in synchronous and asynchronous languages are most often sequentialized and scheduled to be executed as a single thread [Edwards et al., 2006]. Single-threaded implementations of concurrent system level languages have many drawbacks. For example, if the executing thread has to wait for an external event to occur, it blocks the other concurrent behaviors of the program, which do not depend on that event at all. This becomes an even bigger bottleneck if the computation contains heavy data-driven parts. Also, a single thread cannot take any advantage of underlying multiple processors.

4.2.3 Library-based approaches

libGALS is not the first attempt at providing a library-based approach to implement concurrent systems. There are a number of other libraries such as TReK [Gruian et al., 2006], JESTER [Antonotti et al., 2000], JUNIOR [Hazard et al., 1999] and SUGARCUBES [Boussinot & Susini, 1998], which provide support for concurrency. JESTER implements the synchronous MoC, while TReK supports the GALS MoC. Both these approaches rely on a Java Virtual Machine (JVM) and may have low execution speed. They also lack support for important reactive constructs. For example, JESTER does not support deterministic concurrent-exception mechanisms (parallel trap-exit statements), while TReK does not support strong signal-based pre-emptions like abort and suspend. JUNIOR and its derivative SUGARCUBES both follow a completely different semantics [Boussinot et al., 1999]. The JUNIOR reactive kernel implements non-deterministic concurrency, which can lead to undefined behaviours, a problem for mission-critical systems. SUGARCUBES implements logical parallelism, which is mapped to a single threaded implementation.

The libGALS approach combines library and run-time (OS) approach and indirectly supports the language-based approach. It has sequential threads as its basic concurrent units, which are managed by a host operating system that allows the designer to specify concurrent behaviors in a much safer way and guarantee a formal relationship between those behaviors.

The synchronous behaviors can also communicate with each other or with their environment using signals as in synchronous programming languages [Berry, 1993][Boussinot et al., 1999]. Synchronous reactions are implemented as threads in a libGALS program. Behaviors in conventional synchronous programs are sequentialized hence only one behavior is performed at a time. In contrast, reaction threads in libGALS execute concurrently and synchronize with each other at lock-steps according to the GALS MoC. Concurrent behaviors in libGALS programs are mapped to threads supported by the OS following the GALS MoC, and run in true parallel fashion when the underlying platforms allow. Execution times are thus shortened with increasing processor utilization.

4.3 libGALS fundamentals

In this section the concepts and model of computation (MoC) of the libGALS are introduced. Four basic building blocks provided by libGALS, those of clock domain (CD), reaction, signal, and channel, are provided to the designer to construct GALS systems. The concept of logical time (tick) which is used within clock domains is detailed in this section.

4.3.1 Model of computation of libGALS

libGALS extends sequential programming language based on concurrent GALS MoC. The terminology related to GALS is adopted from that used in SystemJ language, because libGALS uses the same semantics described in [Malik, 2010]. A program which utilizes libGALS to model GALS systems is referred to as a libGALS program. Four entities are defined in libGALS: clock domain, reaction, signal, and channel. At the top level, a libGALS program is a composition of one or more asynchronous concurrent entities, which are called clock domains. Communication between clock domains is implemented using channels similar to CSP [Hoare, 1978]. Reactions are behaviors within one clock domain and are synchronous to each other. Synchronous reactions follow the same semantics as ESTEREL [Berry, 2000] and synchronous part of SystemJ [Malik, 2010]. That is, communication between reactions within one clock domain is via signals.

4.3.2 Clock domain: top-level synchronous entity

A clock domain is a top-level entity in a libGALS program. A clock domain itself consists of one or more synchronous behaviors called reactions. Inter-clock domain communications, which occur between reactions belonging to two different clock domains, are implemented using channels. Clock domains execute asynchronously to other clock domains, i.e., at their own logical clocks whose unit is called a ‘tick’. Clock domains are containers where reactions reside. Functionalities of clock domains are defined only in reactions, not in clock domains.

4.3.3 Reaction: behavior of a clock domain

Each reaction can be a composition of further reactions, thus allowing synchronous and hierarchical behavioral concurrency. Reactions are implemented as ‘threads’ which can be created by using the API provided by the underlying operating system. Besides using any of the usual sequential programming language constructs, reactions are also allowed to use a number of control and reactive statements which are available in libGALS. Control and reactive statements enable communication between reactions, as well as with the external environment.

Reactions in the same clock domain are executed in lock-step and are synchronized by a logical tick. Reactions react to environment inputs simultaneously and instantaneously. Outputs are computed and emitted in zero logical time (instantaneously). The reactions of different clock domains communicate with each other through the use of channels, which will be detailed in Section 4.3.5 and 4.5.8. Reactions on each side of the channel work on different copies of the message.

4.3.4 Logical tick in libGALS

A logical tick (different from a tick in the host OS kernel) is used to represent a discrete time instant for a clock domain and all its synchronous reactions, where reactions in each clock domain are executed at its own logical tick. Management of execution of the reactions within each clock domain, and communication with the external environment, are carried out by a helping thread named ‘Synchronizer’ (see Section 4.5.3). The time between two logical ticks, unlike that between two real clock ticks, has variable duration. The tick boundary is determined by various libGALS API calls such as ‘pause’, ‘await’, ‘sustain’, and ‘suspend’. The usage of libGALS API will be detailed in Section 4.4.1.

4.3.5 Signals and traps for communication and synchronization

Signals are the main communication primitives between reactions within clock domains. Communications between reactions and their external environment are also made via signals. Signals can be divided into two major categories: (1) interface signals, used for communication between reactions and the environment and (2) local signals, used for broadcast-based communication between reactions. Signals can be further divided into ‘pure’ and ‘valued’ signals. Pure signals have only a Boolean status (present or absent). Valued signals are a composition of a Boolean status and a value, which can be of any type (void pointers are used in the current implementation). The status of pure and valued signals can be altered with signal emission, which is achieved by calling ‘emit’. The status can be checked using functions like ‘present’ and ‘await’. Similar to reactive languages [Boussinot & Dabrowski, 2006], absence of the signals can be detected only in the next tick. The value of valued signals is persistent over ticks and can be checked via calling ‘value’. Traps are a special kind of signal, used to monitor a specified scope within a reaction body. When executions of reactions are not in the scopes of traps, these traps are not effective. Status of traps can be:

1. Monitoring. Execution of a reaction is still within the scope of a trap. The scope of a trap is bounded by the ‘setTrap’ and ‘endTrap’ calls.

2. Activated to exit. Similar to signal emission, a trap is exited through ‘exitTrap’ call.
3. Not valid. The execution is out of a trap’s scope and the trap is no longer effective.

4.3.6 Channels: communication between clock domains

Channels are the only means of communication between reactions belonging to different clock domains. Channels are point-to-point, unidirectional, and use rendezvous, i.e. blocking send and receive, to guarantee data delivery between reactions. A sender reaction uses the send function and the receiver waits for the data using the receive function. Channels in libGALS operate similarly to CSP of [Hoare, 1978], the sending and receiving sides working on different copies of the message. Invisible delays occur between input and output in the form of empty ticks while waiting for rendezvous in the CSP MoC. In each empty tick, the ‘send’ or ‘receive’ call only ‘pause’, invisible to the programmer at that instance. Empty ticks enable clock domains to still carry out ticks when reactions within are waiting for the channel communication.

4.3.7 libGALS and other software components

Figure 4.1 illustrates relationships between libGALS and other software processes. libGALS is a library implemented at the top of the host OS and requires host OS services including: (1) thread creation and deletion, and (2) semaphore manipulation, which are all available in almost any OS.

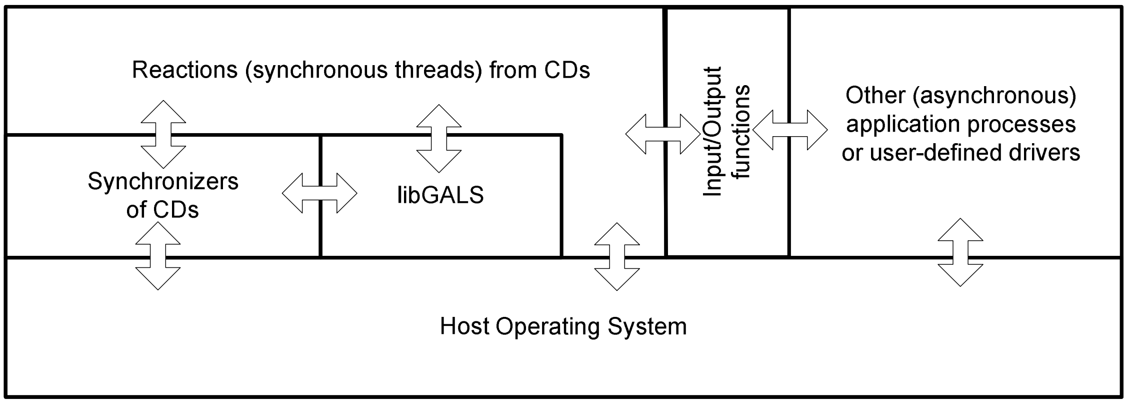

Reactions (synchronous threads) from CDs
Input/Output functions
Other (asynchronous) application processes or user-defined drivers
Synchronizers of CDs
libGALS
Host Operating System

Figure 4.1: libGALS and other software component

Reactions and Synchronizers are implemented as OS threads. Reactions communicate with other application processes and user-defined drivers through input and output functions of the underlying clock domains. Input and output functions can be implemented using inter-process communication (IPC) of the host OS.

4.4 Specifying a design with libGALS

In this section the application programming interface (API) of libGALS is presented. libGALS API is used to construct libGALS programs. An example of alibGALS program, a kite controller used for wind and water surfing, is modeled by using the provided API. The kite controller will be used in a later chapter to demonstrate the linkage between the internals of libGALS with libGALS programs.

4.4.1 libGALS API and libGALS programs

The designer commences the design by dividing the concurrent behaviors into reactions and clock domains. Reactions can then be decomposed into further (child) reactions. The reactions are defined as usual the C functions with a few restrictions which include: (1) use signals instead of shared variables to prevent the use of semaphores, thus avoiding possibility of deadlock and (2) make temporal infinite loops by using at least one statement (function) that consumes logical ticks, i.e. ‘pause’ and ‘await’. The body of the reaction function consists of computational and reactive statements. Computational statements are those of the host programming language (C/C++ in this case), while reactive statements are specified by the libGALS API calls. The comprehensive list of reactive statements with short explanations of their functionality is shown in Table 4.1.

libGALS API calls are categorized into three groups for (1) construction of GALS systems, (2) modeling synchronous behaviors within reactions, and (3) asynchronous communication between clock domains. Groups of API calls are shaded to show differentiation in Table 4.1.

Table 4.1: Application programming interface of libGALS

API	Description
createlibGALSprogram	Initialize a libGALS program
createClockDomain	Create a clock domain
createReaction	Create a reaction within a clock domain
create[Signal | Trap]	Create an instance of a signal or a trap
createChannel	Create a channel connecting two clock domains
startClockDomain	Activate running a clock domain
startlibGALSprogram	Start libGALS program and activated clock domains
initReaction/ endinitReaction	Initialize a reaction and end initialization of the reaction
getArgument	Get an argument passed to the reaction
register[Emitter|Trap]	Register a process as a signal emitter or a trap thrower
emit | sustain	Emit/broadcast (or sustain) a signal
present	Check if a signal is present
pause	Enforce end of tick for a reaction
await	Wait for the presence of a signal
[strong|weak] abort/endAbort	Start and end of a pre-emption block. Pre-empt if monitored signals are present
suspend/endSuspend	Suspend a reaction by one tick if a monitored signals are present
setTrap/endTrap	Set and end the scope of the trap
exitTrap	Exit the trap, the reaction will jump to the end of the trap scope
fork/join	Fork out child reactions and wait for joining of the child reactions
AND,OR,NOT,REP	Form a combined signal expression from presences of signals: AND: logical AND OR: logical OR NOT: logical NOT REP: will return true when a signal emission occurs n times consecutively
value	Acquire the value of a signal
pre[Value]	Get the presence status and value of a signal in the previous tick
endReaction	End a reaction, called if the reaction is not a child reaction
send/receive	Send and receive data between reactions in different clock domains via a channel

API calls in the first group are used to initialize a libGALS program and to create essential compartments of a libGALS program. Clock domains, reactions, signals, traps (special type of signal), and channels are created via this kind of API calls. libGALS program and created clock domains are activated through calling this group of API calls also. Synchronous reactions are described using the second group of the API calls that act as reactive statements. Finally, channel communications between the reactions of clock domains, ‘send’ and ‘receive’ are used as the asynchronous group of the libGALS API calls.

4.4.2 Kite controller: an example of a libGALS program

libGALS enables modular design and re-usability of code in describing GALS systems. For example, the code definition of a reaction, also known as ‘reaction function’, can be used to implement multiple numbers of the actual instantiated *reaction threads*. Signals and channels used in the reaction functions are mapped to actual instances when a reaction is created. A power-kite controller is depicted in Figure 4.2 and its equivalent libGALS program is presented in Listings Listing 4.1 and Listing 4.2, respectively. The power-kite controller consists of three clock domains, which include ‘CDKiteControl’, ‘CDGetWindInfo’, and ‘CDGetKiteInfo’. Speed and heading of the wind and the kite are collected using sensors running at different sampling rates (hence the different clock domains). Collected samples are passed to the ‘rReceiveWindData’ and ‘rReceiveKiteData’ reactions running in parallel synchronously within the clock domain CDKiteControl through channels ‘cWind’ and ‘cKite’, res*p*ectively. This clock domain computes the value of the output signals that control the kite heading and speed, based on this received data. Once calculated, the computed values are emitted via signals to the actuators that stabilize the power kite. libGALS also enables designers to specify test-benches that generate stimuli for testing and validation of the designed system. For example, reactions ‘rSimulateWindData’ and ‘rSimulateKiteData’ generate stimuli that behave as input signals from the environment.

A libGALS program consists of definitions of reactions and a description of the system, which are shown in Listings Listing 4.1 and Listing 4.2, respectively. The definitions of reactions (Listing 4.1) include the definition of user-typed data (lines 2-6) used as arguments in the reactions, the clone function of the user-typed data (lines 7-13), and the body of the reaction functions (lines 14-76). Data sent in both channels cWind and cKite are user-defined type called ‘measurements’, which consists of two components, the heading angle and the speed. The ‘clone function’ measurements_clone is used to duplicate the user-typed data for channel communications to work on different copies of the messages. Lines 14-40 of Listing 4.1 demonstrate how a reaction is defined. A reaction function ‘KiteControl’ is defined with the ‘REACTION_FUNCTION’ macro (line 14). The body of a reaction is divided into two parts, the initializations and the behavior of the reaction. The initialization of the reaction starts with the API call ‘initReaction’ and ends with the API call ‘endInitReaction’ as shown on lines 16 and 30, respectively. Within the scope of the initialization, the arguments passed to create a reaction can be extracted by calling ‘getArgument’ (lines 18 to 25). Signals that will be emitted by this reaction are registered by calling ‘registerEmitter’ (line 28) in the initialization phase. Variables used in the reaction can also be declared in the initialization scope. The behavioral description of a reaction is written after the ‘endInitReaction’ API call. The reaction’s behavior consists of the control part and computational part (data-driven transformations), which are tightly integrated with each other (lines 31-45). Control parts of the reaction are modeled with the libGALS API calls, while computational parts are expressed in the host programming language. To illustrate the hierarchical design in libGALS programs, KiteControl forks out and then waits for joining of child-reactions ‘rReceiveKiteData’ and ‘rReceiveWindData’ with fork and join API calls (lines 32-33). The ‘fork’ and ‘join’ API calls together coordinate the synchronous concurrency model within a clock domain. A reaction can initialize multiple synchronous reactions (called child reactions) concurrently using the ‘fork’ API calls, which instantiate the child reactions. Once the child reactions are initialized the parent waits for their completion before proceeding further. This is done by calling the blocking ‘join’ API call. Computational parts are carried out to determine whether the bearing and speed of the kite need to be increases or reduces. Emission/broadcasting of the signal ‘sIncreaseKiteVelocity’ with an ‘emit’ API call (line 41) is performed to maintain the course of the kite.

Asynchronous communications through channels are carried out with ‘send’ and ‘receive’ API calls on lines 71 and 54, respectively. Both send and receive calls require the name of channel, the data to transfer, and the type of the data. In this case, data typed measurements is used. The behavioral description of a reaction ends with the ‘endReaction’ API call (such as line 46).

The GALS system (Listing 4.2) instantiates the clock domains using the ‘createClockDomain’ API calls (lines 5-10). These API calls take the input and output functions that act on the interface input and output signals as arguments. Input and output functions allow the inputs to the clock domain to be read at the beginning of every tick and the output signals to be emitted at the end of every tick. ‘createChannel’ is used to instantiate channels for communication between the sending and receiving clock domains, along with the name of the channel, which are arguments to this function (lines 11-16). Signals used within the clock domain are created by calling ‘createSignal’ whose argument is the clock domain where the signal operates (lines 18-29). The reactions are instantiated via the ‘createReaction’ API call. The required arguments to create a reaction include:

1. The clock domain where the reaction resides; each created reaction acts synchronously with other reactions created in the same clock domain.
2. The reaction function which hooks with this reaction instance. Each reaction is associated with a reaction function. More than one reaction can refer to the same reaction functions but with no shared context.
3. The activation status of the reaction (activated or dormant); an activated status is of value 1 and 0 otherwise. Child reactions are dormant before being forked from the parent reaction. For instance, ‘rReceiveWindData’ and ‘rReceiveKiteData’ are dormant (lines 33 and 40)) and wait for activation from ‘rKiteControl’**,** which is activated initially (line 47).
4. The number of arguments passed to the reaction function (lines 34, 41, 48, 61, 69, 76, 83, and 90).
5. The actual arguments are provided as arguments. For instance, the creation of reaction ‘rKiteControl’ (line 44) indicates the reaction will be active upon the creation. Furthermore, eight arguments will be passed to the reactions, which include two child reactions, four output signals, and two input channels.

Note that two instances of the reaction function ‘ReadData’ are created on line 59 and 88, which demonstrates the modularity and code re-usability provided by libGALS. The clock domains are started using the ‘startClockDomain’ API calls (lines 96-98). Finally the GALS system starts with the ‘startlibGALSProgram’ (line 99). Synchronizers are programmer-invisible threads to manage activities of each clock domain.

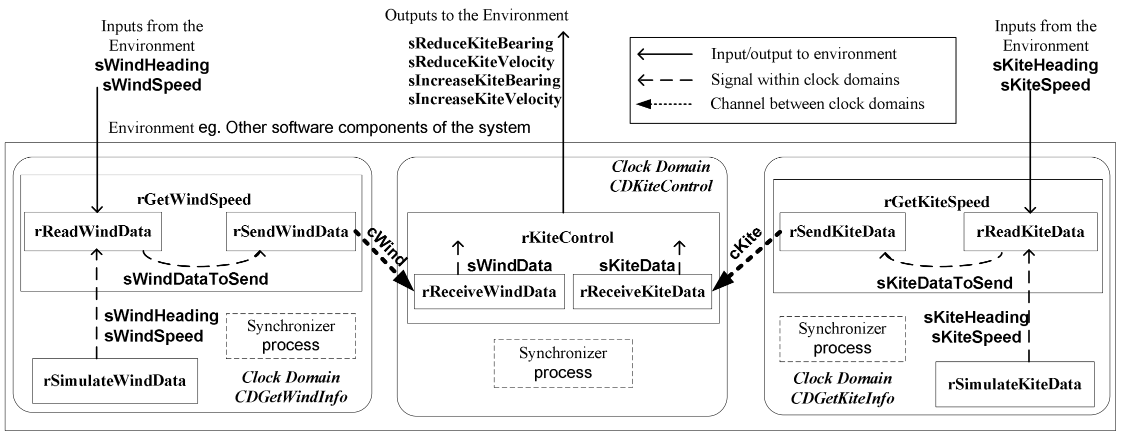


Figure 4.2: Power kite control system abstract representation

Listing 4.1: Definition of reaction functions

```
1      #include “libGALS.h”
2      typedef struct measurements {
3      // Definition of User Types
4        int heading;
5        int speed;
6      } measurements;
7      measurements* measurements_clone(measurements* original) {
8        measurements * newMeasurements =
9            malloc(sizeof(measurements));
10       newMeasurements->heading = original->heading;
11       newMeasurements->speed = original->speed;
12       return newMeasurements;
13     }
14     REACTION_FUNCTION(KiteControl) {
15       // Initialize data structure used by the reaction
16       initReaction();
17       // Obtain arguments passed to this reaction
18       reaction rReceiveKiteData = (reaction)getArgument(1);
19       reaction rReceiveWindData = (reaction)getArgument(2);
20       signal sIncreaseKiteBearing = (signal)getArgument(3);
```

```
    signal sIncreaseKiteVelocity = (signal)getArgument(4);
    signal sReduceKiteBearing = (signal)getArgument(5);
    signal sReduceKiteVelocity = (signal)getArgument(6);
    signal sWindData = (signal)getArgument(7);
    signal sKiteData = (signal)getArgument(8);
    .......
    // Register the output signals of this reaction
    registerEmitter(IncreaseKiteVelocity);
    ...... // Declare variable used within the reaction
    endInitReaction();
    while(1) {
      fork(rReceiveKiteData); fork(rReceiveWindData);
      join(rReceiveKiteData); join(rReceiveWindData);
      crossWind = sin(sWindData->heading-sKiteData->heading)*
                        sWindData->speed;
      // cross_wind within limits
      if(abs(CrossWind) < MAX_CROSS_WIND) {
        headwind = cos(sWindData->heading-sKiteData->heading)*
                        sWindData->speed;
        if (headWind>0&& head_wind < MAX_HEAD_WIND){
          emit(sIncreaseKiteVelocity, 0);
        }
        ......
      ......
    }
    endReaction();
  }
  REACTION_FUNCTION(ReceiveData) {
    initReaction();
    channel cData = (channel)getArgument(1);
    signal sData = (signal)getArgument(2);
    ......
    ChannelDataType *data;
    receive(cData, data, ChannelDataType);
    emit(sData, data);
    ......
  }
  REACTION_FUNCTION(GetSpeed) { ...... }
  REACTION_FUNCTION(ReadSpeed) { ...... }
  REACTION_FUNCTION(SendData) {
    initReaction();
    signal sData = (signal)getArgument(1);
    channel cData = (channel)getArgument(2);
    ......
    // Await and store sData to headingData and speedData
    ...
    ChannelDataType *data =(ChannelDataType*)malloc(
        Sizeof(ChannelDataType));
    data->heading = headingData;
    data->speed = speedData;
    send(cData, data, ChannelDataType);
    ......
  }
  // TestBench Reaction
```

```
REACTION_FUNCTION(SimulateKiteData) { ...... }
REACTION_FUNCTION(SimulateWindData) { ...... }
```

Listing 4.2: Definition of the GALS system

```
#include “libGALS.h”
#include “ReactionFunctions.h”
int main(void) {
  createlibGALSProgram();
  clockdomain CDKiteControl = createClockDomain(InputC0,
                                                OutputC0);
  clockdomain CDGetWindInfo = createClockDomain(InputC1,
                                                OutputC1);
  clockdomain CDGetKiteInfo = createClockDomain(InputC2,
                                                OutputC2);
  channel cWind = createChannel(CDGetWindInfo,
                                CDKiteControl,
                                “cWind”);
  channel cKite = createChannel(CDGetKiteInfo,
                                CDKiteControl,
                                “cKite”);
  // Signals for clock domain CDKiteControl
  signal sIncreaseKiteBearing = createSignal(CDKiteControl);
  signal sIncreaseKiteVelocity = createSignal(CDKiteControl);
  signal sReduceKiteBearing = createSignal(CDKiteControl);
  signal sReduceKiteVelocity = createSignal(CDKiteControl);
  signal sWindData = createSignal(CDKiteControl);
  signal sKiteData = createSignal(CDKiteControl);
  // Signals for clock domain CDGetWindInfo
  signal sWindHeading = createSignal(CDGetWindInfo);
  signal sWindSpeed = createSignal(CDGetWindInfo);
  signal sWindDataToSend = createSignal(CDGetWindInfo);
  // Signals for clock domain CDGetKiteInfo
  ......
  reaction rReceiveKiteData = createReaction(
      CDKiteControl,    // Clock domain that reaction is in
      ReceiveData,  // Reaction function
      0,                // Set Active status to dormant
      2,                // Number of argument(s)
      cKite,            // Pass cKite as the argument
      sKiteData);       // Used to pass data to rKiteControl
  reaction rReceiveWindData = createReaction(
      CDKiteControl,
      ReceiveData,
      0,
      2,
      cWind,
      sWindData);
  reaction rKiteControl = createReaction(
      CDKiteControl,
      KiteControl,
      1,
      8,
      rReceiveKiteData,
      rReceiveWindData,
            sIncreaseKiteBearing,
            sIncreaseKiteVelocity,
            sReduceKiteBearing,
            sReduceKiteVelocity,
            sWindData,
            sKiteData);
        reaction rReadWindData = createReaction(
            CDGetWindInfo,
            ReadData,
            1,
            3,
            sWindSpeed,
            sWindHeading,
            sWindDataToSend);
        reaction rSendWindData = createReaction(
            CDGetWindInfo,
            SendData,
            1,
            2,
            sWindDataToSend,
            cWind);
        reaction rGatherWindData = createReaction(
            CDGetWindInfo,
            GatherData,
            1,
            2,
            rReadWindData,
            rSendWindData);
        reaction rSimulateWindData = createReaction(
            CDGetWindInfo,
            SimulateWindData,
            1,
            2,
            sWindSpeed,
            sWindHeading);
        reaction rReadKiteData = createReaction(
            CDGetKiteInfo,
            ReadData,
            1,
            4,
            sKiteSpeed,
            sKiteHeading,
            sKiteDataToSend,
            kiteSamplingPeriod);
        ...... // creation of other reactions
        startClockDomain(CDKiteControl);
        startClockDomain(CDGetWindInfo);
        startClockDomain(CDGetKiteInfo);
        startlibGALSProgram();
    }
```

4.5 libGALS internals

Internal representations of clock domain, reaction, signal, and channel are detailed in this section. Concepts such as the helping thread and the scheduling policy used to govern internals of libGALS are described in the following text.

4.5.1 Overview of the libGALS data structure

Each libGALS program works on a programmer-invisible data structure, which is illustrated in Figure 4.3. API calls within reactions operate on the underlying data structure. Information within such data structure is used to book-keep status of the libGALS program.

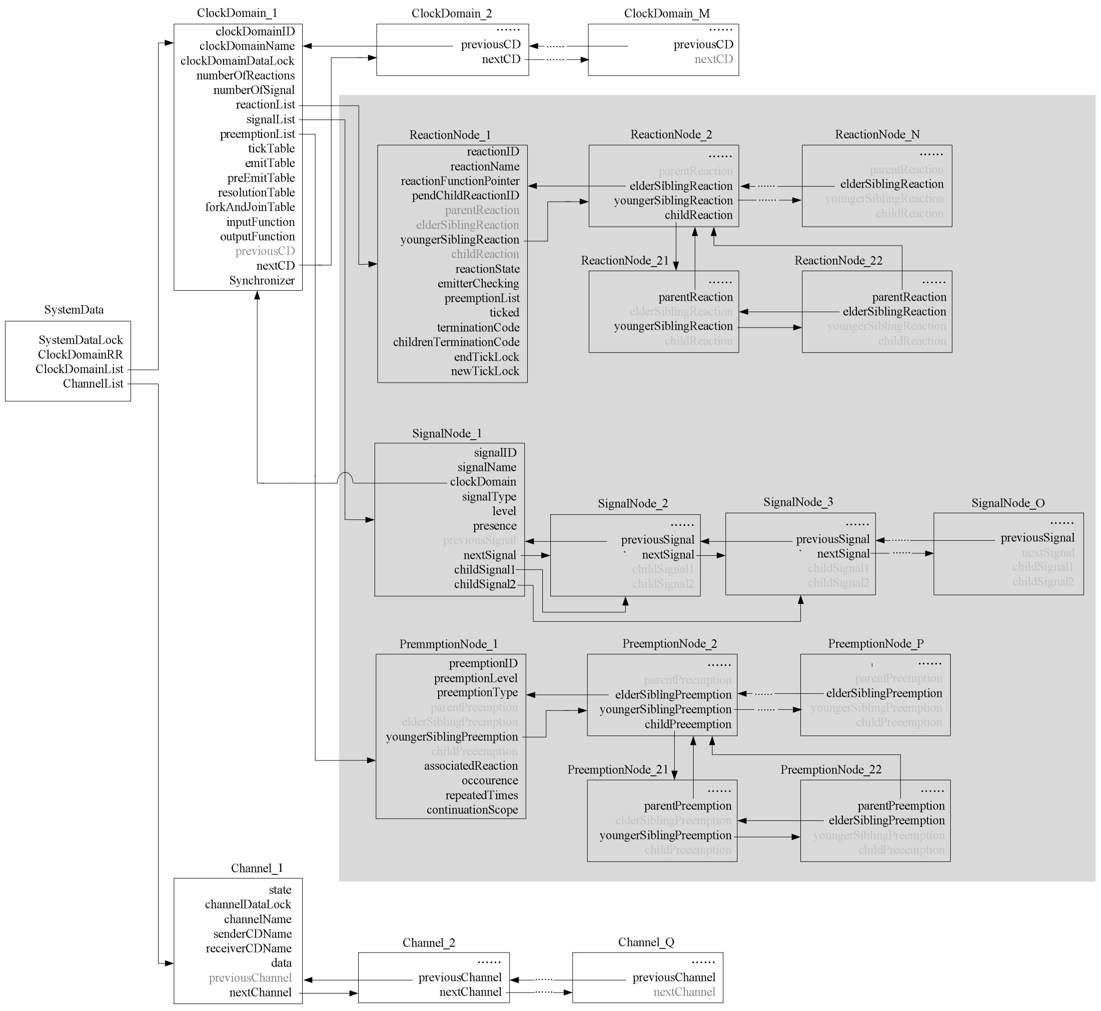


Figure 4.3: Data structure of a libGALS program

'SystemData' holds a list of clock domains and channels in the 'globally asynchronous' realm. Clock domains and channels are stored in link-lists and *SystemDataLock* is a semaphore-typed lock to ensure data consistency when adding clock domains and channels to a libGALS program. ClockDomainRR is used only when the underlying OS does not provide suitable scheduling policy to prevent starvation of clock domains.

Data structures of reactions, signals, and pre-emptions of different clock domains are managed independently and shown as shaded in Figure 4.3. They are linked with the corresponding clock domain data structure. Traps use the data structure of a signal since traps are special cases of signals.

4.5.2 Clock-domain data structure

Each clock domain operates on its own data structure whose fields are listed in Table 4.2. Notice that each clock domain has a 'clockDomainDataLock' which is similar to SystemDataLock of SystemData, for data integrity within each clock domain.

Table 4.2: Fields of ClockDomain data structure

Field Name	**Description**
clockDomainID	The ID of the clock domain. The ID is issued based on the order of clock domain creation
clockDomainName	The name of the clock domain. It is used to identify sending and receiving sides of a channel
clockDomainDataLock	This is used to ensure the data consistency within the clock domain when calling API which operates on reactions and signals of the underlying clock domain
numberOfReactions	The number of reactions in the clock domain
numberOfSignal	The number of signals in the clock domain
reactionList	A pointer to a link list of data structure ReactionNode. The list contains the information of reactions resided in the clock domain
signalList	Similar to reactionList, signalList is a pointer to a link list of data structure SignalNode representing the list of used signals in the clock domain
preemptionList	A list of the PreemptionNodes, which is monitored in the current tick. Pre-emptions are used at the beginning and end of ticks to perform strong and weak pre-emptions
tickTable	Records the tick status of each reaction in the clock domain

emitTable/ preEmitTable	To record the status and values of the signals in the current tick and previous tick
resolutionTable	A table whose rows and columns are equal to NumberOfReaction and NumberOfSignal respectively. This table is used to dynamically resolve signal dependencies at run-time
forkAndJoinTable	A table whose rows and columns are equal to NumberOfReaction. It is used to maintain fork-and-join activities between parent and child reactions
inputFunction/ outputFunction	These are two function pointers pointing to user-defined functions to communicate with the environment
previousCD/nextCD	Points to the previous and next instance of clock domain in the libGALS program
Synchronizer	Points to a Synchronizer function, which manages ticks and signal resolutions in a clock domain

4.5.3 Synchronizers

Synchronizers are helping threads within a libGALS program. A synchronizer is created whenever a new clock domain is created. The services provided by Synchronizer are: (1) dynamic resolution of signal dependencies, (2) synchronization of reactions at the clock-domain tick boundaries, (3) maintenance of internal data structures for the new tick, such as book-keeping of the previous status and values of signals, (4) call of the input and output functions to communicate with the environment and (5) update of channels' status to implement rendezvous between reactions belonging to different clock domains. The implementation of Synchronizer is simply an infinite loop, which provides services when all the other reactions in the same clock domain are blocked. In priority based OSs, such as μCOS-II, Synchronizers is implemented as the thread with the lowest priority, compared to reaction threads, to prevent taking up control of the processor.

4.5.4 Reaction internals

Synchronous reactions are implemented as threads whose execution bodies are defined as reaction functions. 'ReactionNode' is used as the data structure to represent the status of a reaction and to relate the reaction to the other components within a clock domain, such as pre-emption and signals. The fields of ‘ReactionNode’ are shown in Table 4.3. The behavior of a reaction is described by a finite state machine illustrated in Figure 4.4. State transitions are resulted from libGALS API calls or actions of the Synchronizer and are listed in Figure 4.4. A tick of a reaction can span over one or more FSM states depending on the interaction with other reactions.

Table 4.3: Feilds of ReactionNode data structure

Field	**Description**
reactionID	The ID of the reaction
reactionName	The name of the reaction
reactionFunctionPointer	Points to reaction function as the execution body
pendChildReactionID	The reaction ID of the child reaction which is awaited by the parent reaction, it is used by the join API call
parentReaction	Pointer to the parent reaction, if there is any
elderSiblingReaction/ youngerSiblingReaction	Pointers to reactions that share the same parent reaction
childReaction	Pointer to the first child reaction if there is any
reactionState	An enumeration showing the current reaction state, as shown in Figure 4.4.
emitterChecking	It is used for signal resolutions
preemptionList	Pointer to the innermost pre-emption scope in the reaction. The underlying pre-emption scope is also located in the list of pre-emptions of the resident clock domain
ticked	Indicate if a reaction has finished its tick
terminationCode	The status of the reaction. It is set to 1 at the end of tick, values greater than 1 if the reaction is ended due to pre-emption, -1 if it is checking the presence of the signal, and 0 if it is not active or running computational statements or non-blocking libGALS API
childrenTerminationCode	Present the returning status of the child reactions. This structure guarantees that the current reaction will proceed further with execution only once all the children have finished executing
endTickLock	It is used to signal Synchronizer that the end of tick of the reaction has reached
newTickLock	It is signaled by Synchronizer to information the reaction to start a new tick

Forking of a reaction will establish interconnect between the parent and child reactions. Figure 4.5 illustrates one of the relationships of the nodes in the power kite controller shown in Figure 4.2. Note that ‘childReaction’ of rKiteControl points to the first child (rReceiveKiteData) that it forks out. Sibling relationships are formed between the rReceiveKiteData and rReceiveWindData reactions. reactionFunctionPointer of both rReceiveKiteData and rReceiveWindData refer to reaction function ReceiveData.

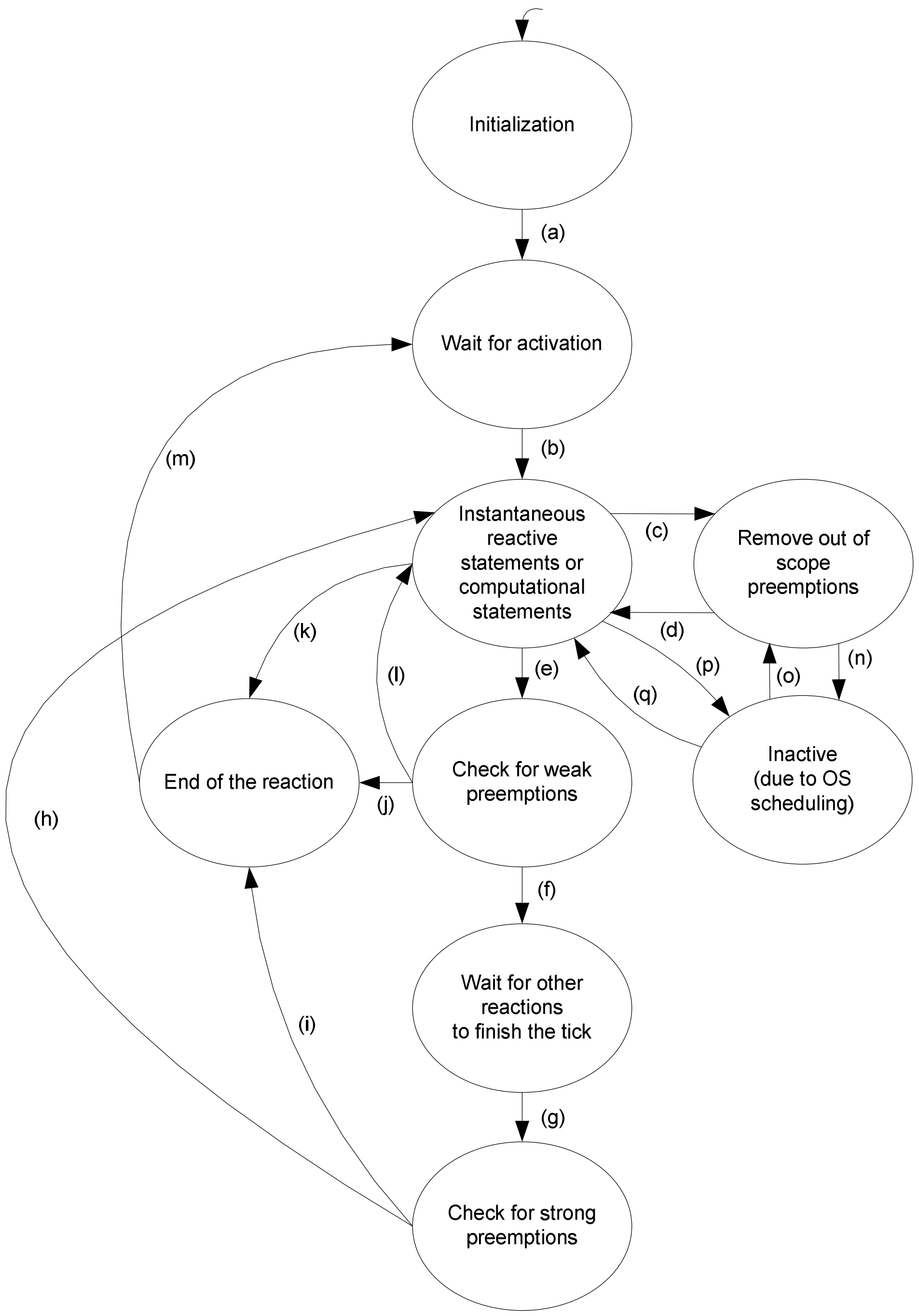

Initialization
(a)
Wait for activation
(b)
(m)
Instantaneous reactive statements or computational statements
(c)
Remove out of scope preemptions
(d)
(k)
(l)
(e)
(p)
(q)
(o)
(n)
End of the reaction
Check for weak preemptions
Inactive (due to OS scheduling)
(j)
(h)
(f)
Wait for other reactions to finish the tick
(i)
(g)
Check for strong preemptions

Figure 4.4: Finite state machine of a reaction

Table 4.4: State transition of a reaction

State transition	Description
a	End of reaction initializations
b	Forked by parent reaction or activated while created
c	libGALS API such as endAbort indicates the end of a pre-emption scope
d	Pre-emption scope removed, return from libGALS API
e	Encounter tick boundary, eg a ‘pause’ call
f	No weak pre-emption is activated
g	All active reactions reach the end of tick
h	Start new tick, jump to continuation address if strong pre-emption is activated
i	Strong pre-emption set by parent reaction is activated
j	Weak pre-emption set by parent reaction is activated
k	Reach end of the reaction or join the parent reaction
l	Weak pre-emption activated
m	Wait to be activated by a parent reaction
n	Blocking caused by the operating system service requested by libGALS when processing API call
o	Unblocking from the previous blocked libGALS API call
p	Transfer control to other reactions or processes due to the scheduling policy of underlying operating system
q	Control of the processor transferred from other reactions or processes due to operating system scheduling

Forking and joining are managed through forkAndJoinTable as shown in Figure 4.6. It is a two-dimensional structure where each row-to-column element is directly mapped to a parent-to-child reaction relationship. For instance, the element located at the intersection of the first row, second column, indicates that the first reaction of the clock domain is the parent reaction to the second reaction of the clock domain. Each element is a node and contains two binary semaphores. These semaphores are used for activation and resumption of the parent and child threads. Both the number of rows and columns of the table structure are equal to the number of reaction threads. To conserve the run-time memory, memory for each element is only allocated when there is a ‘fork’ call. The allocation of the element is freed once the corresponding parent and child reaction threads finish the fork and join phases.

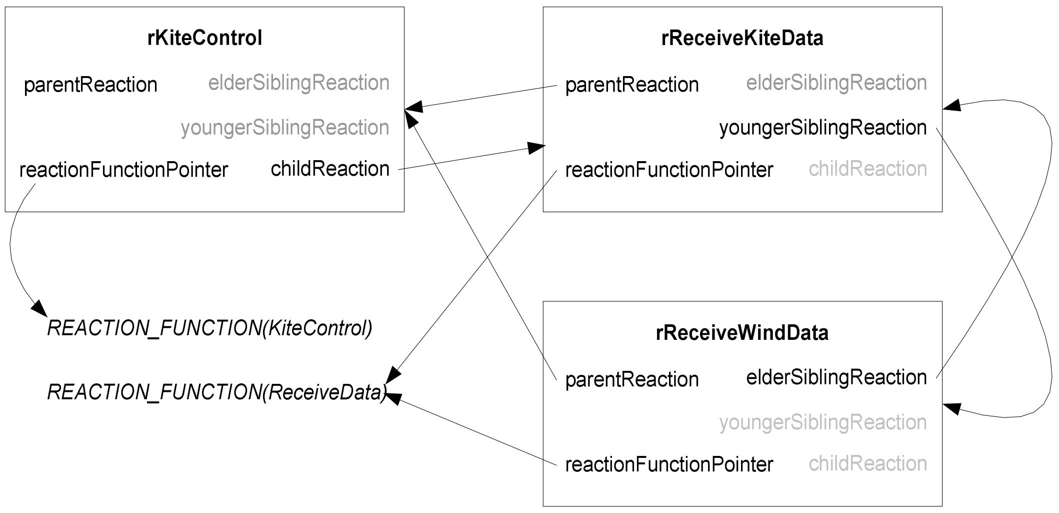


Figure 4.5: Interconnection of ReactionNodes after forking

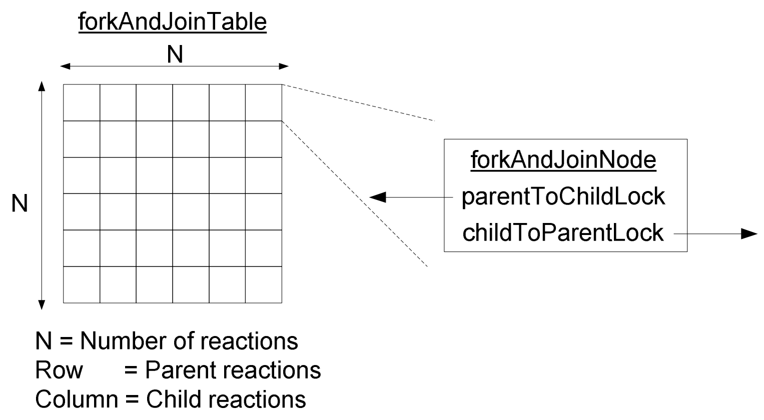


Figure 4.6: Data structures used to achieve fork and join of reactions

4.5.5 Scheduling of reactions within clock domains

The scheduling of reactions is handled by the host OS scheduler. This scheduling mechanism works closely with Synchronizers and the internal data structures of a libGALS program. If a reaction is blocked due to a libGALS API call, control is transferred to another reaction that is ready for execution. The interleaving of reaction execution and transfer of control from one reaction to another are governed by the scheduling policy of the host OS. For instance, the libGALS Linux implementation adopts the use of POSIX threads and, the scheduling decisions are thus made by the Linux scheduler. However, the reaction cannot be scheduled unless it has the permission of its clock domain Synchronizer, which enforces lock-step execution of reactions.

Scheduling strategies on different operating systems affect only the execution sequence of reactions which do not have mutual signal dependencies. A reaction in one clock domain can be executed in parallel with reactions in other clock domains if the execution platform allows it (for example on a multiprocessor or multicore platform). Figure 4.7 illustrates an example of three reaction threads where Reaction 2 and Reaction 3 depend on the emission of signal A, and the sequence of execution is such that Reaction Thread 1 takes the first step. Once the signal A is emitted, all three reactions can run in parallel depending on the number of available processing units.

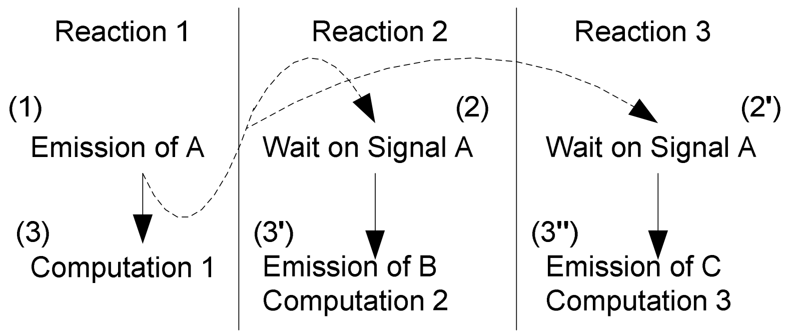


Figure 4.7: True parallelism of reaction threads on multiprocessing cores

Another example of libGALS implementation is in the embedded operating system, µCOS-II [Labrosse, 2002], which features pre-emptive scheduling. Scheduling strategies on different operating systems will affect only the execution sequence of independent micro steps of reaction threads. Dependencies such as checking on presence of signals and conditions of pre-emptions are handled by the libGALS library and the behavior of the reactive program is still deterministic.

4.5.6 Signal representation and resolution

Signals in the libGALS are represented by ‘SignalNode’ data structure. A SignalNode is positioned in signalList of the clock domain that utilizes such signal. Fields in SignalNode are listed in Table 4.5.

Table 4.5: Fields of SignalNode data structure

Field	Description
signalID	The ID of the signal
signalName	The name of the signal
clockDomain	The clock domain that the signal is created in
signalType	The type of the signal: 0: signal created by using createSignal API call 1: signal created by using AND 2: signal created by using OR 3: signal created by using NOT 4: signal created by using REP
level	The level of the signal starts from 1 indicating the SignalNode is created by using createSignal API call. Otherwise the SignalNode is created by other means.
presence	The status of the signal
previousSignal/ nextSignal	Used by signalList of the underlying clock domain. Point to the previous and next signalNode in the list
childSignal1/ childSignal2	Point to SignalNodes which are used as arguments of singal combination API calls such as AND and OR

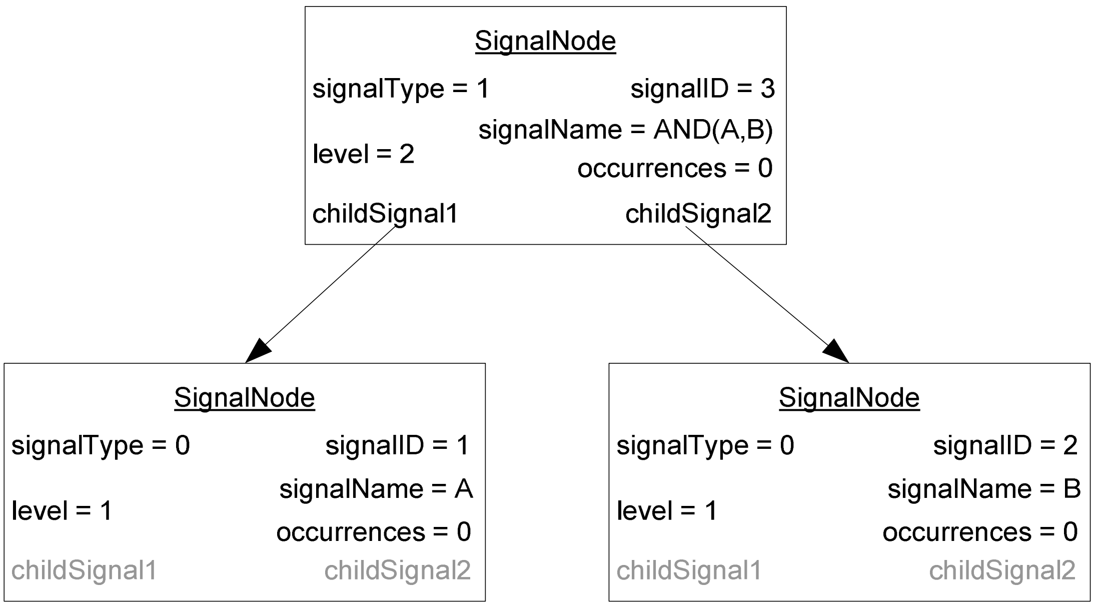


Figure 4.8: Interconnection of signal nodes

A signal node (implemented by using SignalNode data structure) can represent a single signal or an operation on a signal (such as NOT), or logical combinations of signals (such as AND between two signal status). A SignalNode is created when corresponding libGALS API calls are made. For instance, *AND(A,B)* will operate over three SignalNodes (one created for AND, two existing for signals A and B) and will form an ‘and’ relationship of signals A and B. The interconnections between the signal nodes are illustrated in Figure 4.8.

API call AND(A,B) established a SignalNode whose SignalID is 3. childSignal1 and childSignal2 of such SignalNode point to signals A and B, respectively. Both signals A and B are created by calling createSignal, therefore levels of these signals are 1 with signalType of 0. Subsequently, SignalNode AND(A,B) are assigned with level 2 and signalType 1.

The presence of a signal (or their logical combination) also determines the dependencies between reactions. For instance, a ‘present’ statement in one reaction cannot proceed until the signal, which is checked for presence, is emitted or ruled out by control flow in this logical tick, otherwise the ‘present’ will execute the wrong control branch. A ‘resolutionTable’ is created in each clock domain to comply with this signal broadcast MoC as detailed in Figure 4.9. Each element of the resolution table, called *resolutionNode*, indicates the relationship between a signal and a reaction in the clock domain. Fields of resolutionNode are detailed in Table 4.6. Synchronizer carries out the resolution process of a signal according to the internal status of the resolutionNodes. Synchronizer has a global view of the resolution table, where dependencies can be detected and resolved. Example of strategies of signal resolutions include but are not limited to:

1. Resolve a signal if the emitter reaction thread has finished its tick.
2. Resolve a signal if the emitter reaction thread is not active and does not wait for the joining of any child-reaction threads.

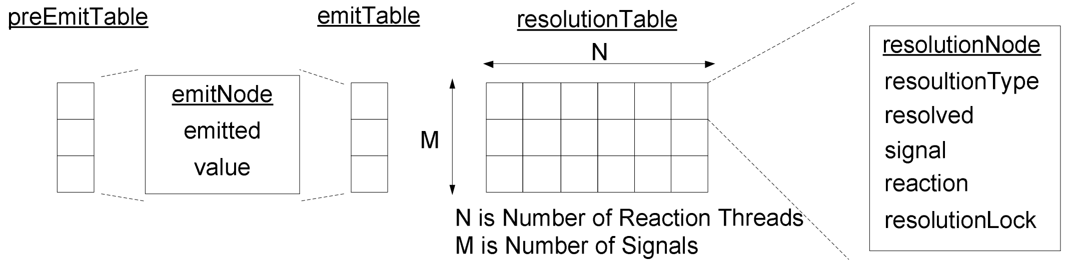


Figure 4.9: Data structures used to resolve signals

The ‘emitTable’ created in each clock domain stores the status (emitted member of the emitNode structure) and the values of signals. A signal is identified as emitted when it is fully resolved and has been emitted by one of the reaction threads, which are the emitters. The ‘preEmitTable’ stores the status and values of signals in the previous tick, which is used by the ‘pre’ and ‘preValue’ API calls.

Figure 4.10, Figure 4.11, and Figure 4.12 illustrate how signals are resolved in the scenario presented in Figure 4.7. In each resolutionNode, status such as resolutionType, resolved, and resolutionLock are shown. Firstly, reactions 2 and 3 are blocked due to checking the presence of signal A as shown in Figure 4.10. resolutionLocks of blocked reactions are in a pending status. Signal A is then emitted (shown in emitTable) and resolved (resolved = 3 for signal A, in resolutionTable), which leads to the releases of resolutionLocks, as illustrated in Figure 4.11. Reactions 2 and 3 continue to be executed. Finally, signals B and C are emitted in Figure 4.12.

Table 4.6: Fields of resolutioNode

Field	**Description**
resolutionType	Indicates the relationship between the signal and the reaction: 0: No relationship 1: The reaction is currently blocked due to checking the presence of the signal 2: The reaction has been registered as an emitter of the signal
resolved	The presence of the signal has been resolved or not
signal	Pointer to the SignalNode
reaction	Pointer to the ReactionNode
resolutionLock	A lock to block execution of the reaction. It is pending when resolutionType is 1 and is signaled to release when the signal is resolved

Exits of traps are implemented in a manner similar to signal emissions. However, the available operation on traps is limited to checking the status of the trap. Traps share the resolution table with signals.

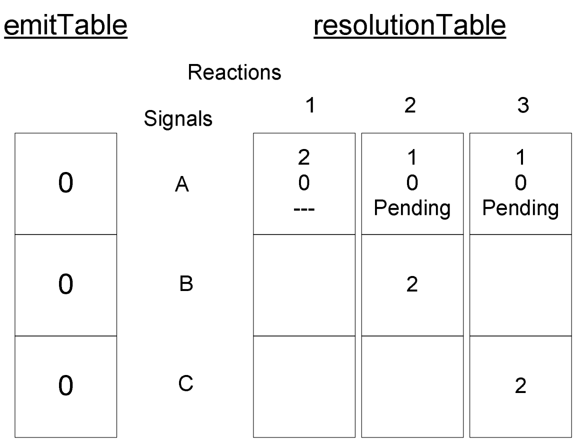


Figure 4.10: Reactions 2 and 3 are blocked

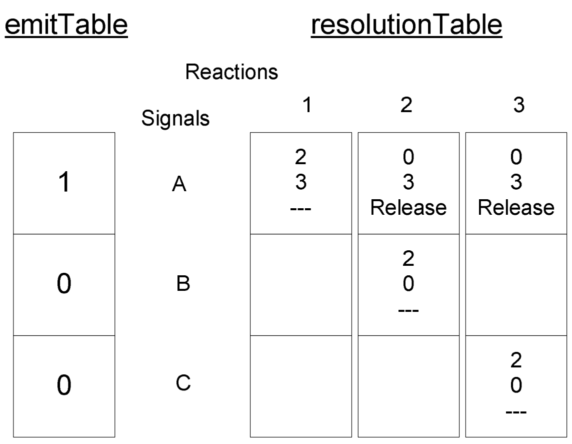


Figure 4.11: Signal A is emitted and reactions 2 and 3 are released

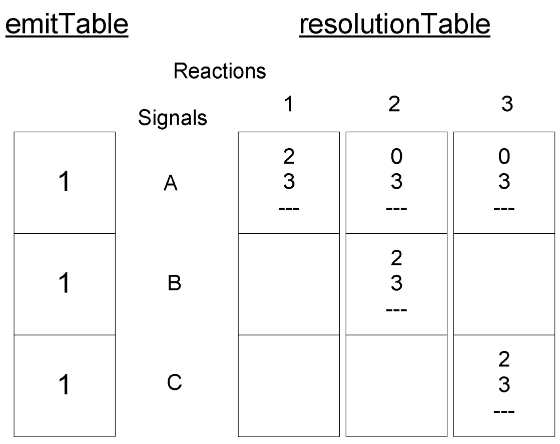


Figure 4.12: Signal B and C are emitted

4.5.7 Pre-emption representation and activation

A pre-emption scope in a reaction is represented by a data structure called ‘PreemptionNode’ illustrated in Figure 4.13. Figure 4.14 details the interconnections of the PreemptionNode and ReactionNode resulting from Listing 4.3. When a reaction enters a scope of a monitored pre-emption, such as ‘StrongAbort’, the PreemptionNode of the corresponding pre-emption scope is created. When a monitored PreemptionNode is detected to be active, the execution will be pre-empted from the tick boundary and carried out from the ‘continuationScope’, which is the end of the underlying pre-emption scope. The member ‘preemptionList’ of a reaction points to the innermost PreemptionNode resident in the reaction. Note that the innermost pre-emption scope will take a lower precedence than the outer pre-emption scope(s). Nested pre-emption scopes are assigned with different ‘preemptionLevels’.

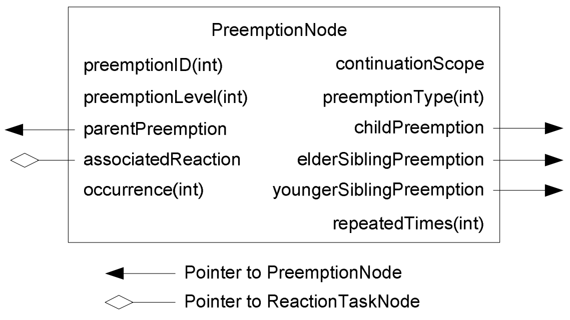


Figure 4.13: Preemption Node

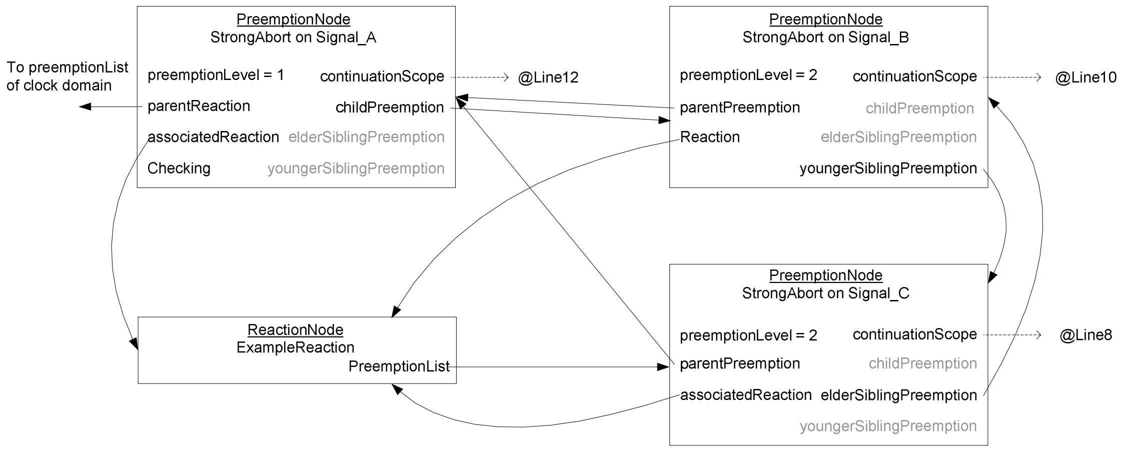


Figure 4.14: Relationships between pre-emption nodes and reaction thread node

Note that ‘traps’ and ‘suspends’ are special cases of pre-emptions. Traps are similar to weak pre-emptions (aborts). The PreemptionNode of a trap is activated when ‘exitTrap’ is called. ‘suspend’ is similar to strong abort. Instead of redirecting the reaction thread to the continuationScope, a tick is delayed when the condition of suspension is true. Thus, the PreemptionNode of a suspend statement lacks the continuationScope.

Listing 4.3: Nested pre-emptions

```
void ExampleReaction(void *data) {
  ...... // other statements
  StrongAbort(Signal_A, AbortName1) {
    StrongAbort(Signal_B, AbortName2) {
      StrongAbort(Signal_C, AbortName3) {
        ...... // Other statements
      }
      EndAbort(AbortName3);
    }
    EndAbort(AbortName2);
  }
  EndAbort(AbortName1);
}
```

4.5.8 Channel communication internals

A channel data structure is created when ‘createChannel’ is called. Both sending and receiving reactions of different clock domains operate on the same instance of Channel. Fields of Channel are detailed in Table 4.7. The ‘state’ variable of Channel is used to identify the status of the data transfer. Figure 4.15 illustrates a Moore-type finite state machine of a channel. The states and the transitions of states are described in Table 4.8 and Table 4.9, respectively.

For each user-typed data used in channel communication, a clone function is required. The name of the clone function for data-typed ‘dataTyped’ is in the form of ‘dataTyped_clone’. The input argument is the original data and the output of the function is the new instance of the data which is a duplication of the original. The output of the clone function is then pointed to by the ‘data’ field in Channel. This enables channel communications to work on different copies of the messages. The clone function is called along with the ‘send’ API call.

Table 4.7: Fields of channel

Field	Description
state	The state of the channel. Details of states are listed in Table 4.8
channelDataLock	This is used to keep the consistency of the channel data, because channel data are accessed by two reaction threads, that is, the sending and receiving reactions
channelName	The name of the channel
senderCDName	The clock domain name where send is called
receiverCDName	The clock domain name where receive is called
data	A pointer points to the duplicated version of the original data for receiver to read
previousChannel/ nextChannel	Point to the previous and next instance channel in the libGALS program

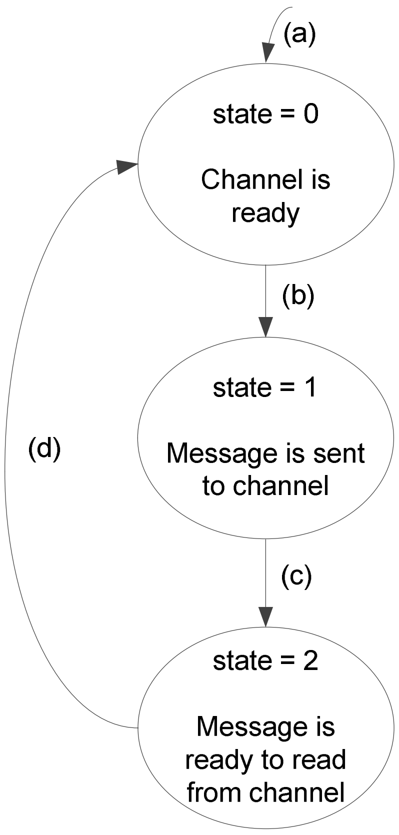

(a)
state = 0
Channel is ready
(b)
state = 1
Message is sent to channel
(c)
state = 2
Message is ready to read from channel
(d)

Figure 4.15: Finite state machine of channel communications

Table 4.8: States of a channel

States	Description of activates	
	Sending side	**Receiving side**
0	Channel is ready	Channel is ready
1	Calling clone function to duplicate original data. Assign result to 'data' field of the channel	
2		Read data field of the channel and return to the *receive* call

Table 4.9: State transitions of a channel

State transitions	Description
a	'creatingChannel' is called
b	'send' is called
c	'receive' is called
d	Returned from the receive call, received data is assigned to the destination

4.6 Applications and ports of libGALS

4.6.1 Mapping GALS/synchronous models to libGALS programs

As libGALS provides all mechanisms to implement the GALS MoC, it also gives the opportunity to implement existing GALS and synchronous languages using concurrent processes.

SystemJ [Malik, 2010] is a GALS language which can be implemented by using libGALS. The SystemJ statements can be directly compiled onto libGALS API calls. Examples of a few mappings are provided in Table 4.10.

Table 4.10: Examples of mapping from SystemJ to libGALS

SystemJ Statements	Mappings with libGALS
present S { … }	Present(S) { … }
emit S;	emit(S);
pause;	pause();

abort (S) { … }	strongAbort(S, AbortName) {
	  ……
	}
	endAbort(S, AbortName);

4.6.2 Porting libGALS

libGALS has been ported to general operating systems such as Linux and Windows via POSIX interface, where the pthread library is used in implementation. Both of these operating systems offer scheduling mechanisms to provide high fairness between processes and threads, and hence the high response times. Reactions in libGALS do not necessarily require special care to change attributes of the mapped threads/processes, such as priority.

Ports of embedded and real-time operating systems (RTOS) are similarly available. Since libGALS requires only features such as task creation/deletion and semaphore, effort in porting libGALS to different operating systems is minimal. Existing libGALS ports on RTOS include eCos [Massa, 2003], RTEMS[RTEMS, 2003], FreeRTOS[Barry, 2008], and µCOS-II [Labrosse, 2002]. Since eCos and RTEMS provide POSIX interface and cooperative scheduling policy, they are very close to the Linux port of libGALS. FreeRTOS and µCOS-II provide sufficient APIs for libGALS implementation, and the used API calls are listed in Table 4.11. Note that the semaphore mechanism in FreeRTOS is based on message queue, and because of the lack of semaphore deletion API call, *vQueueDelete* is known as the function to call to delete the created semaphore.

Table 4.11: APIs used to implemented libGALS

Operating system features	**POSIX based**	**FreeRTOS**	**µCOS-II**
Task creation	pthread_create	xTaskCreate	OSTaskCreate
Task deletion	pthread_exit	vTaskDelete	OSTaskDelete
Semaphore type	sem_t*	xSemaphoreHandle	OS_EVENT
Semaphore pending	sem_wait	xSemaphoreTake	OSSemPend
Semaphore signaling	sem_post	xSemaphoreGive	OSSemPost
Semaphore creation	sem_init	xSemaphoreCreateCounting	OSSemCreate
Semaphore deletion	sem_destroy	vQueueDelete	OSSemDel

Because µCOS-II does not allow multiple tasks with the same priority, reactions of clock domains are divided into different priority groups. This leads to the issue that one of the clock domains may have monopoly over processor time and not ever give control to the other reactions of other clock domains. This is resolved by introducing member

'ClockDomainRR' in SystemData data structure mentioned in section 4.5.1. ClockDomainRR is implemented as a counting semaphore that forces clock domains to take their turns of execution or be scheduled in a specific ratio of executions.

The Synchronizer task of a clock domain can be seen as the lowest priority task, providing services when all the reaction threads of the clock domain are blocked.

4.7 Experiments and results

In order to demonstrate performance of the libGALS programs, they are compared with SystemJ programs that implement the same functionality, since SystemJ is practically the only GALS language with an available compiler. All examples are with mixed data-driven and control-driven operations. A frequency relay (FR) has been used as an example and is illustrated in Figure 4.16.

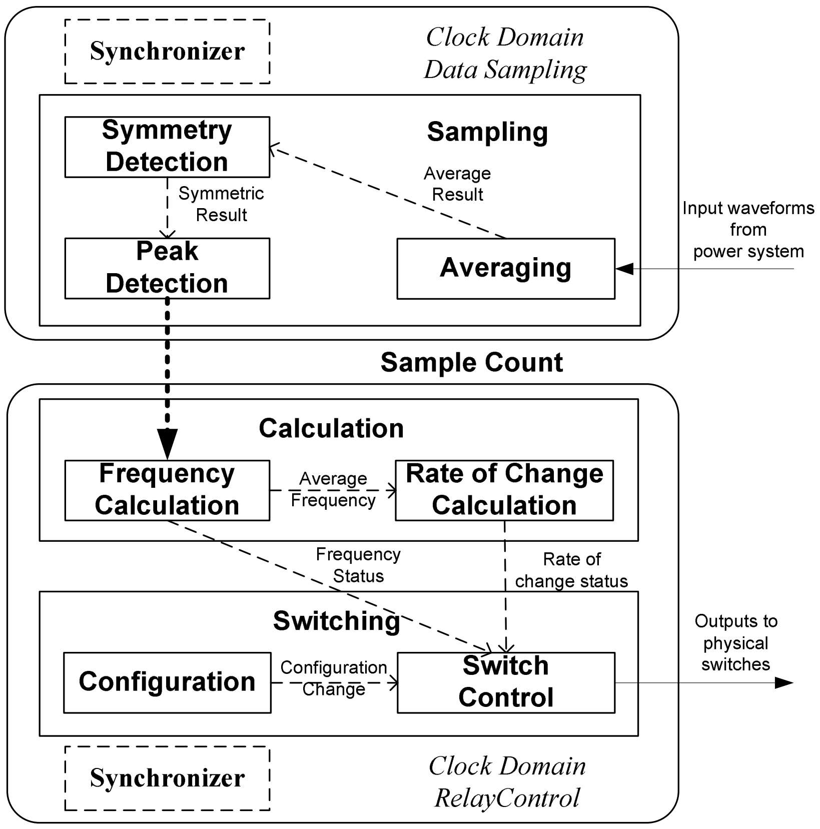


Figure 4.16: Frequency relay implemented as a GALS system with two clock domains

Frequency relay consists of two major parts, data sampling and relay control [Salcic & Mikhael, 2000]. In data sampling, signal processing algorithms are performed. Sampled power signal waveform is processed in an averaging filter by using a moving window concept. It is followed by the symmetry function calculation to simplify the procedure of finding peak points of the waveform instead of zero-crossings, carried out later in the peak detection function. Time periods between peaks are obtained to allow calculation of frequencies. The rate of change of frequency is also computed. To maintain a stable power network, working frequencies and rate of change of frequency must be within a specified range. If they are out of range, the loads will be shed from the network. This is carried out by the switching facilities and relay control part.

The frequency relay is partitioned to two clock domains, ‘DataSampling’ and ‘RelayControl’. The DataSampling clock domain consists of four reactions: the parent reaction, reaction ‘Sampling’, forks out reaction ‘Averaging’, reaction ‘Symmetry detection’ and reaction ‘Peak detection’. Clock domain RelayControl is a composition of two larger reactions: reaction ‘Calculation’ and reaction ‘Switching’. Both reactions have two child-reactions. ‘Frequency calculation’ and ‘Rate of change calculation’ are child reactions of reaction Calculation, delivering essential information to ‘Switch control’ reaction under reaction ‘Switching’ to perform load shedding, if necessary. Reaction ‘Configuration’ is the other child reaction of reaction Switching, which provides parameters of the frequency relay to reaction Switch control. These two clock domains communicate through ‘SampleCount' channel.

Table 4.12: Comparisons between SystemJ and libGALS

Example	Average tick time (μs)		Code Size (Bytes)	
	libGALS	SystemJ	libGALS	SystemJ
2CD Frequency Relay	27.67	75.23	33,865	101,469
2CD KiteController	11.37	27.16	9,431	59,296
2CD Async Proto	48.37	16.25	13,078	52,800
2CD Data Comp	18.23	26.37	865	10,920
3CD Data Comp	17.72	39.28	975	11,944
4CD Data Comp	17.43	56.62	1,085	13,010

*Note that the code size of libGALS is 33K Bytes.

SystemJ examples are compiled with the latest SystemJ compiler to generate single-threaded Java source code, which is compiled by the Java compiler version 6.0 and then run on a JVM. The equivalent libGALS examples are compiled with gcc-4.3.1. Experiments were carried out on Intel Core 2 Quad 2.4GHz with 4GB of RAM with Linux 2.6.29.6 as the host OS. Results are shown in Table 4.12.

The libGALS approach consistently results in smaller object code size, because the single-threaded SystemJ code emulates both synchronous- and asynchronous-concurrency with switch-case statements. On the other hand, libGALS implements concurrency with threads. Note that the code size of the SystemJ implementation does not include the code size of the JVM, which is larger than the standard C run-time library. Execution speed has been compared through an average-tick execution-time of one million ticks. The libGALS approach shows advantages if the data computations are heavier. The ‘3CD Data Comp’ and ‘4CD Data Comp’ consist of three and four clock domains, respectively. In these cases libGALS takes advantage of multicore processing. SystemJ is advantageous if clock domains are highly control-dominated as in the 2CD Async Proto example.

4.8 Summary

In this chapter a run-time library approach, libGALS, for extension of the sequential programming language (C/C++, for instance) to enable specification of GALS concurrent systems is proposed. libGALS provides an application programming interface (API) that enables the designer to describe GALS programs in these sequential programming languages. This enables efficient integration of control-driven and data-driven components of a design.

The approach is based on the features of a host OS, made available to the programmer via a set of API. Programs designed with libGALS comply with the GALS MoC and thus provide a much safer programming approach compared with the use of traditional threading libraries. libGALS implements GALS concurrency by using multiple processes or threads, unlike the current system-level languages that compile the specification into a single-threaded code. This not only improves responsiveness of the resulting programs, but also offers the advantage of executing such programs on multiprocessor and multicore systems. Because of this, libGALS opens a new path towards the compilation of GALS languages, as well as of synchronous languages as their subset. The other advantage of libGALS programs is their ability to interface with other tasks and drivers in the host with minimal effort. This allows major future development, targeting the dynamic creation of clock domains, synchronous reactions and whole GALS programs, thus supporting software system run-time adaptation and reconfiguration, as will be described in Chapter 6.

5

GALS-Designer: A design framework for GALS software systems

GALS-Designer is a framework for the design of software systems which comply with formal globally asynchronous locally synchronous model of computation (GALS). The framework integrates the libGALS library for writing libGALS programs and SYSTEMC. In Chapter 4, a library called libGALS to model GALS systems as libGALS programs has been introduced. GALS systems may consist of single or multiple libGALS programs and their immediate environment, which can be other programs and any other modules described in SYSTEMC. It enables modeling and simulation of single and multiple libGALS programs within the single SYSTEMC executable model on the host (simulation) operating system. The same libGALS programs then can be run without SYSTEMC on a target operating system for which the libGALS run-time library is available.

The use of the GALS-Designer is demonstrated on an example of a complex embedded system. As libGALS can ultilize multiprocessor platforms, both simulation and target models of the GALS system can take advantage of multiprocessor and multicore systems, which is not possible when using standard SYSTEMC. Results of running simulation models of libGALS programs demonstrate simulation performance improvement when performing on multicore platforms.

5.1 Introduction

In this chapter the GALS-Designer, the marriage between libGALS and SYSTEMC in the single design framework, is presented. GALS-Designer enables the modeling of complex systems that include hardware and other concurrent components, e.g. models of the physical world and the environment, along with software-system components that are represented by libGALS programs. In the proposed approach, libGALS is used to specify libGALS programs, which are then wrapped into SYSTEMC modules and can be simulated together with other SYSTEMC modules within the same SYSTEMC execution model. Simulation of such a multicomponent system can be carried out with different timing granularities, depending on the current development phase of the overall system, so the designer can use trade-offs between faster simulation and more accurate timing behavior of the system. libGALS programs, once simulated within a SYSTEMC model can be translated to the implementation code which will be executed on a target operating system. SYSTEMC is chosen as the basis because of its ability to (1) model hardware, software and environment of the designed system with different levels of abstraction and timing granularity, (2) result in hardware and software synthesis, (3) cooperate with models made in other languages which can be linked with the SYSTEMC library to obtain the host simulating executable and (4) model the interaction with the environment, thus effectively providing test benches, which is essential for validation of design through simulation.

This chapter is organized as follows. Section 5.2 presents the related work and positions the contributions. In Section 5.3, principles of the GALS-Designer are introduced. Integration of libGALS and SYSTEMC is given in Section 5.4, followed by the programming model used in GALS-Designer in Section 5.5. Using GALS-Designer in system level design is presented in Section 5.6. A case study and the results of using the proposed approach are given in Section 5.7, followed by a summary of this chapter in Section 5.8.

5.2 Related works and fundamentals

5.2.1 Synchronous and GALS system models

Linkage between SYSTEMC and synchronous languages such as Esterel was explored and presented previously. Brandt and Schneider demonstrated that a set of Esterel programs can be translated to SYSTEMC with certain limitations [Brandt & Schneider, 2008]: (1) programs respond to delayed actions, i.e. signals emitted in the previous clock cycle, and (2) pre-emptions are not modeled. Sun et al. present a case study on how to convert an Esterel program into SYSTEMC description simulated with the abstract RTOS model [Sun & Salcic, 2007]. In [Radojevic et al., 2006], both Esterel and SYSTEMC are used to model systems described in DFCharts. Significant effort is required to manually translate an ESTEREL program to SYSTEMC, with numerous restrictions on the use of SYSTEMC constructs. An automatic generation of SYSTEMC model from COLA is presented in [Wang et al., 2008], where COLA follows the perfect synchrony semantics. However, it produces only a simulation model.

Furthermore, in synchronous languages like Esterel compiler resolves causality problems of signal dependencies, which is not possible in the library-based approach used in SYSTEMC. Other synchronous languages such as SL [Boussinot & De Simone, 1996], JESTER [Antonotti et al., 2000], JUNIOR [Hazard et al., 1999] and SUGARCUBES [Boussinot & Susini, 1997] provide support for concurrency. However they do not support the GALS MoC. TReK [Gruian et al., 2006] and SystemJ [Malik, 2010] provide GALS MoC for software systems, but do not allow simulation of interaction between SystemJ program and other components in the system, particularly those describing hardware. Also, since SystemJ programs require Java virtual machine (JVM), it is not suitable for real-time applications.

5.2.2 Modeling software concurrency with SYSTEMC

Concurrent software is often implemented as a collection of processes, or threads, governed by an operating system. Modeling of (real-time) operating systems, (RT)OS, in SYSTEMC is not new. A summary of modeling strategies is presented in [Posadas et al., 2005]. (RT)OS model often provides information in different timing granularities, from untimed to timed, with different resolutions, especially for the task scheduling. (RT)OS modeling in SYSTEMC can be categorized as follows:

1. Model a target processor in SYSTEMC. The processor will read the executable target binary from the modeled memory. The target binary is obtained by linking concurrent software tasks with OS. The modeled processor (sometimes called ‘emulator’) behaves as the real processor but internal details of the processor are abstracted for the faster simulation speed.
2. Execute target binary on the simulation host through the instruction set simulation (ISS). The ISS either could communicate with the SYSTEMC simulation kernel through inter-process communication (IPC) with the host, or be linked with the SYSTEMC simulation kernel.
3. Software tasks are executed in the ISS, and they interact with the OS model described in SYSTEMC. Communication between the ISS and the OS model follows the previous category. [Krause et al., 2008] demonstrates such kind of modeling strategy.
4. The proprietary OS simulator is provided in a library form. The developer can choose to link the task codes with the OS library and then simulate with SYSTEMC as in point number 2.
5. An OS model described in SYSTEMC provides a set of application programming interface (API), which is the same as that of the original (real) OS, linked with the task codes and SYSTEMC library. The OS model can be from very abstract to very detailed.
6. Both OS and the tasks are modeled in SYSTEMC, communication and synchronization between the tasks and between the tasks and the OS are via SYSTEMC constructs. However, the underlying model of computation might

be different from the original SystemC description to the final embedded software because of different scheduling policies of different targeted OS. Even when support for the OS modeling is once introduced to SystemC version 3, the inconsistency with the MoC will still remain when a different OS is used. Furthermore, the nature of the SystemC simulation kernel determines that only one host process is used to execute the simulation executable regardless of the number of concurrent processes in a SystemC module.

Herrera et al. present how embedded software can be generated from SystemC descriptions through the use of concurrent threads managed by an OS [Herrera et al., 2003]. Few restrictions are set when describing a concurrent software process in SystemC, such as using channels for inter-process communication instead of using shared member-variables in a SystemC module. Inasmuch channels and process management are mapped onto services such as mutex and thread management, provided by the underlying RTOS, they do not follow any formal MoC. Various (RT)OSs behave differently over similar sets of APIs, i.e. the implementation will be different from the simulation model. SoCOS presents a framework to model dynamicity and concurrency of software through the use of C++ [Desmet et al., 2000]. Focusing on simulation it proposes a library-based approach to support the execution of generated software on an OS. Posadas et al. present a POSIX model in SystemC [Posadas et al., 2005] and its implementation with an OS compatible with the POSIX standard; however, it limits the selection of the target OS.

Based on previous work and known constraints, a modeling technique is presented to integrate programs that use GALS MoC, libGALS programs, with SystemC components by using GALS-Designer framework. The major contributions of this approach and work are:

1. It enables a developer to describe a concurrent application software system that complies with the formal GALS MoC, and simulate its execution together with other SystemC components on a host OS. The same libGALS

program can be executed on the target platform OS with almost no modification (the modification is done by a simple text parser that removes simulation-related parts).

2. It enables the use of different timing granularities in simulation models. Details such as execution times can be annotated in dedicated hook-functions which will be introduced in Section 5.5 and 5.6. With designers able to choose between faster simulation and higher accuracy, depending on the requirements, GALS-Designer can be used in different design stages.
3. It supports scalability by enabling the use of multiple libGALS programs with any number of asynchronous behaviors (clock domains), as well as any number of synchronous behaviors inside each of the asynchronous behaviors in the same model, as is explained in more detail in Section 5.3.
4. It enables faster and more efficient simulation by enabling the use of a multithreaded multicore execution platform, not practicable with usual SYSTEMC models.

5.3 Overview of GALS-Designer

5.3.1 Integration of libGALS and SYSTEMC

GALS-Designer is a framework for designing GALS software systems, which may consist of single or multiple libGALS programs. GALS-Designer uses SYSTEMC and libGALS as the backplane for system models. Both SYSTEMC and libGALS are libraries built on top of the C++ and C, respectively, as shown in Figure 5.1 (a). They both provide interfaces to access the library and generate executables with which to be linked. Systems modeled in SYSTEMC can use libGALS to describe libGALS programs as a part of an overall system model. The executable model that combines parts described with libGALS and SYSTEMC runs on the host OS, as shown in Figure 5.1 (b). The execution is started as a SYSTEMC executable, which is governed by the SYSTEMC simulation kernel. When the modeled libGALS programs start executing, threads mapped from reactions are spawned. These threads are managed by the libGALS with the aid of the OS, and are executed concurrently with the SYSTEMC simulation kernel. libGALS programs synchronize and communicate with other hardware and software (HW/SW) components modeled using SYSTEMC.

Once the designer switches from the simulation to the implementation phase, SYSTEMC library is removed, and the translation from libGALS program models to their implementation version is performed. The resulting libGALS program is then linked with the version of the libGALS for the target execution platform and target OS, which may be different from the one used in simulation, but with identical API. This situation is illustrated in Figure 5.1 (c).

SYSTEMC provides to libGALS the necessary modeling mechanisms for the description of the environment in which the libGALS programs will run. This enables modeling of inputs/outputs (such as user-inputs and sensor data), other software components in the system and hardware components communicating with the libGALS programs.

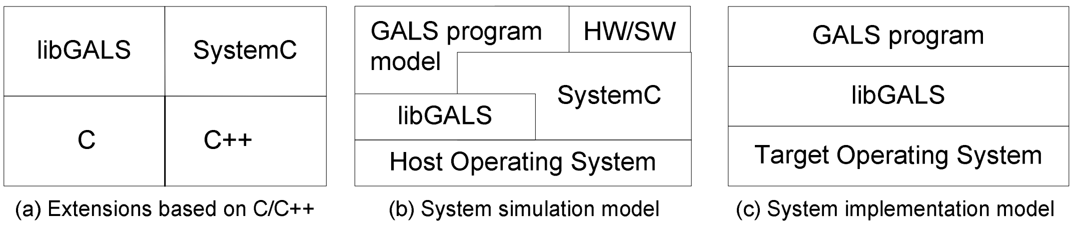


Figure 5.1: Relationships between libGALS and SYSTEMC

Each libGALS program is described and modeled in a single SYSTEMC module. A libGALS program model can communicate with other SYSTEMC modules as to its environment through communication constructs provided by SYSTEMC, as shown in Figure 5.2. libGALS programs can also communicate with each other through modeled channels. These are abstracted and may have different underlying implementations such as sockets and network-communication links, inter-process communication (IPC) mechanism, etc.

As detailed in Chapter 4, each libGALS program consists of a number of asynchronous concurrent behaviors called clock domains, which communicate with each other using rendezvous-based channels. The name clock domain is used to emphasize the fact that it may consist of a number of synchronous concurrent behaviors, called reactions, which execute in lock-step with a logical clock called 'tick' and follows the rules of the synchronous-reactive model of computation [Berry & Gonthier, 1988].

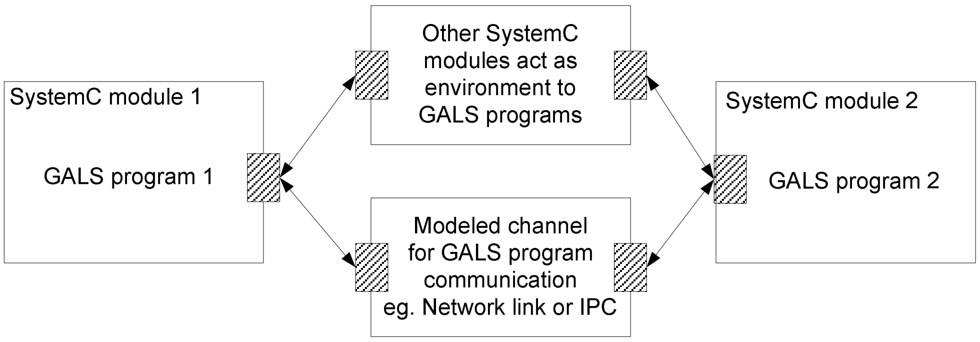

Other SystemC modules act as environment to GALS programs
SystemC module 1
GALS program 1
SystemC module 2
GALS program 2
Modeled channel for GALS program communication eg. Network link or IPC

Figure 5.2: Communications of libGALS program and other SYSTEMC components

5.3.2 Linkage between libGALS programs and SYSTEMC

Figure 5.3 illustrates how libGALS and SYSTEMC mechanisms are used to form a SYSTEMC module representing a libGALS program which is specified by using mechanisms provided within the libGALS. In addition, the libGALS program uses hook functions to communicate with the external environment of the libGALS program, in this case other modules of the SYSTEMC model. A clock domain in a libGALS program communicates with its environment synchronously through a sampling process. This process receives information from the environment either periodically (e.g. using the clock) or by an event-driven pre-emption mechanism (e.g. using interrupts). Synchronizing functions are introduced to model this. Synchronizing functions can be triggered by either external clock (synchronously) or other signals when input data is ready (asynchronously), and are synchronized again with hook functions through 'SyncNodes' when input data is required by the libGALS program. Outputs from the libGALS program to other SYSTEMC modules are implemented using the same concept.

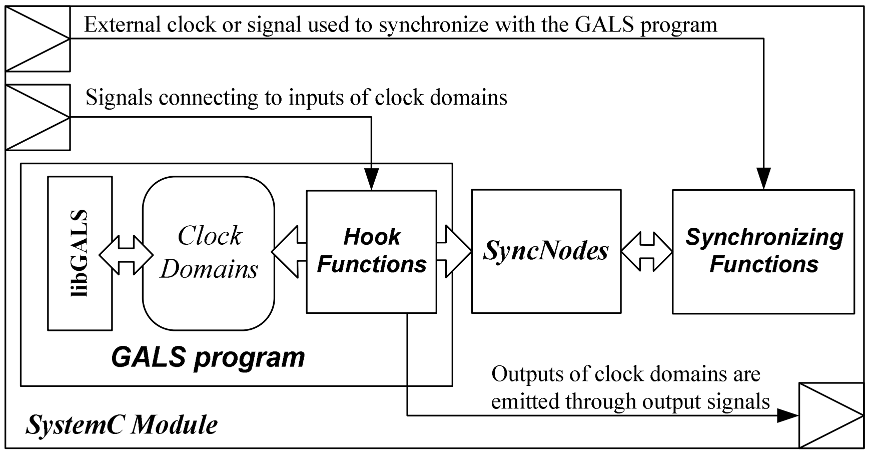

External clock or signal used to synchronize with the GALS program
Signals connecting to inputs of clock domains
libGALS
Clock Domains
Hook Functions
SyncNodes
Synchronizing Functions
GALS program
Outputs of clock domains are emitted through output signals
SystemC Module

Figure 5.3: A SYSTEMC module wrapping a libGALS program model

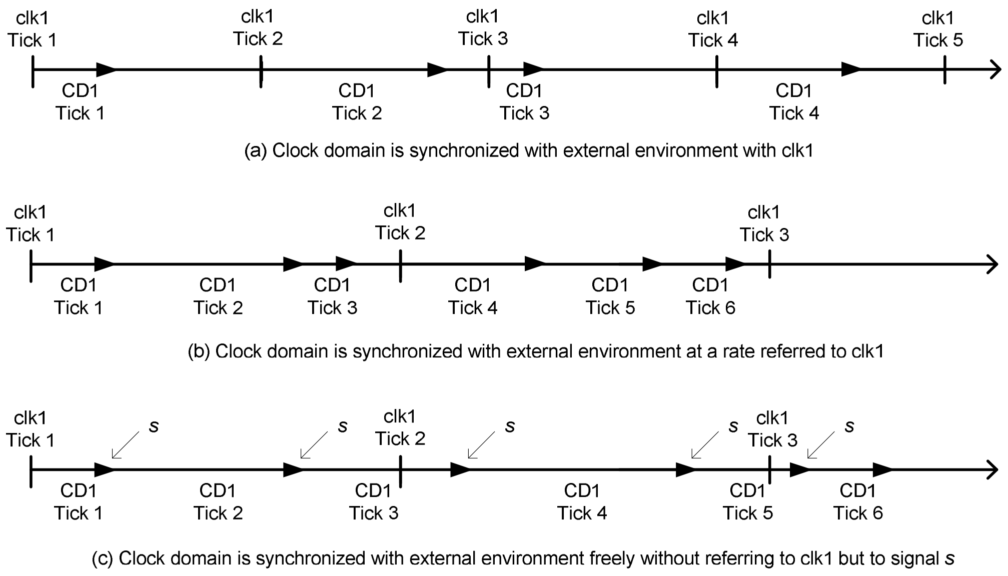


Figure 5.4: Synchronizations between libGALS-SYSTEMC and other SYSTEMC modules

The synchronization between the libGALS program and its environment through SyncNodes takes the following forms (as illustrated in Figure 5.3):

1. A clock domain inside a libGALS program, i.e. CD1 in Figure 5.4 (a), is synchronized with the environment by an external clock clk1. The clock domain finishes its logical tick before the tick of the external clock arrives.
2. Figure 5.4 (b) illustrates how a clock domain synchronizes with the environment in multirate fashion through the external clock. For example,

the hook function of the clock domain synchronizes with the synchronizing functions every three logical clock ticks.

3. A clock domain synchronizes with the environment when signal s is valid to read.
4. Signal s is activated by other SYSTEMC modules as shown in Figure 5.4 (c).

In the next section it is shown how libGALS and SYSTEMC are combined into GALS-Designer, where they collaborate in modeling complex systems.

5.4 Integration of libGALS and SYSTEMC

To enable interoperability and integration of libGALS and SYSTEMC, some aspects need to be addressed:

1. libGALS and SYSTEMC are implemented in C and C++, respectively, which requires resolution of compatibility between the two libraries.
2. GALS programs execute at logical ticks, with different logical clocks for each clock domain, in contrast to SYSTEMC models, which can be simulated at different levels of time granularity. Synchronizations between libGALS programs and SYSTEMC modules need to be established, as discussed in Section 5.3.
3. Since the libGALS program is used not only in simulation but also in the implementation, interfaces provided by libGALS to describe libGALS programs should be preserved in both simulation and implementation. This way the libGALS can be used in different phases of the system-design cycle.

Because libGALS is a C library, in order to use it together with SYSTEMC, which is an extension of C++, programs written in C have to be used in C++ with the ‘extern C { … }’ construct. This construct is utilized in this approach as the glue mechanism between libGALS programs and SYSTEMC descriptions.

Since SYSTEMC is used to model the environment of the libGALS programs, the libGALS program is ‘wrapped’ into a SYSTEMC module that provides interface to other SYSTEMC modules. A SYSTEMC module generally consists of the following:

1. Interfaces of the module.
2. Member variables representing the attributes and the structure of the module.
3. Member functions, which can either be used as concurrent processes, or can be private functions to carry out algorithms. In libGALS programs, clock domains and reactions, channels, and signals are modeled as member variables of the module. Interfaces of the wrapping module are also member variables.

As mentioned previously, libGALS programs are running at the pace of their clock domain logical ticks, which are different to a SYSTEMC simulation clock, and thus synchronization between a libGALS program and other SYSTEMC modules is required. Furthermore, since the input and output functions of a libGALS program operate on a libGALS signal object and are thus not able to access member variables of the wrapping module, a set of member functions to the wrapping module is introduced, called ‘interfacing functions’. As interfacing functions are responsible for the communication and synchronization between the libGALS program and its wrapping SYSTEMC module, they need to be recognized by both. Because interfacing functions are member functions, they can access the member variables such as the module interfaces and signals of the libGALS program. It would, moreover, be inefficient to check whether a tick of the libGALS program has elapsed by using polling, and, even more important, it is also possible to miss a libGALS program ticks, since threads from the libGALS program are running at speeds different from the SYSTEMC simulation. Therefore, interfacing functions must be registered with the libGALS program so that they can be activated when a tick completes.

Interfacing functions are categorized into (1) tick-hook functions and (2) synchronizing functions. Tick-hook functions are registered with clock domains and reactions of a libGALS program, and the synchronizing functions are defined as processes in the wrapping module. The tick-hook functions are non-static in order to enhance the re-usability of the libGALS-SYSTEMC module. It means that if there is more than one instance of the same module, the static functions of all these instances would operate on the same data set, which is impractical and error prone. However, in C++ (hence in SYSTEMC) only static functions are allowed to create the threads/processes which are essential to libGALS programs. Therefore a dedicated static-function wrapper is introduced to wrap each non-static member function to become a static function. When a clock domain or a reaction is created in the wrapping module, the static-function wrapper is passed as the tick-hook function. The static-function wrapper takes an argument, called SyncNode, implemented as a data structure which contains a pointer to the actual tick-hook function.

The SyncNode data structure maintains the link between the hook functions and synchronizing functions. SyncNodes are member variables of the wrapping module and are instantiated when a 'tick-hook and synchronizing functions' pair is required. Figure 5.5 illustrates the SyncNode structure and its operations. A SyncNode consists of (1) a function pointer that points to a tick-hook function of a clock domain or a reaction, (2) synchronization constructs: the current implementation in Linux uses two semaphores from pthread and (3) a set of member functions to perform handshaking. The SyncNode contains two semaphores, which are used by the tick-hook function and the synchronizing function to implement the handshaking.

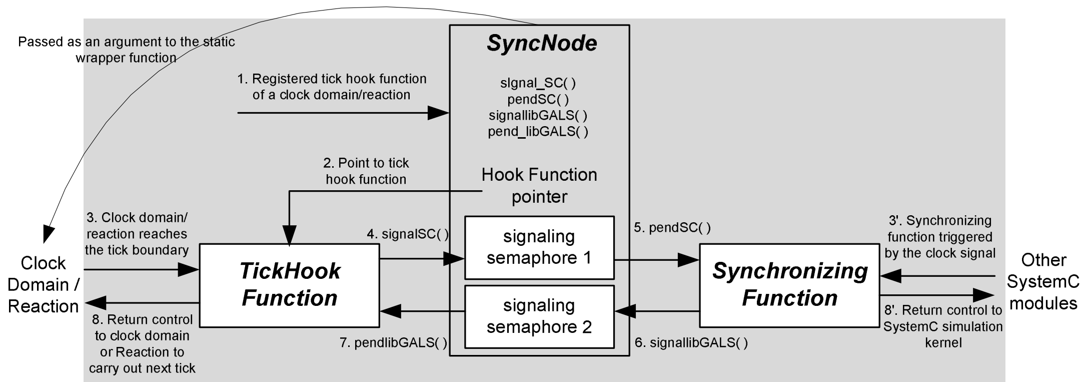

Passed as an argument to the static wrapper function
SyncNode
signal_SC( )
pendSC( )
signallibGALS( )
pend_libGALS( )
1. Registered tick hook function of a clock domain/reaction
2. Point to tick hook function
Hook Function pointer
3. Clock domain/ reaction reaches the tick boundary
4. signalSC( )
5. pendSC( )
3'. Synchronizing function triggered by the clock signal
Clock Domain / Reaction
TickHook Function
signaling semaphore 1
signaling semaphore 2
Synchronizing Function
Other SystemC modules
8. Return control to clock domain or Reaction to carry out next tick
7. pendlibGALS( )
6. signallibGALS( )
8'. Return control to SystemC simulation kernel

Figure 5.5: Synchronization steps between tick hook and synchronizing function

To abstract the details of synchronization during the handshaking, four member functions are introduced to SyncNodes:

1. signalSC – the tick-hook function requests to synchronize
2. pendSC – the synchronizing function is ready to synchronize
3. signallibGALS – the synchronizing function accepts the synchronization
4. pendlibGALS – the synchronization is finished

When a SyncNode is created, a corresponding tick-hook function is first registered with the SyncNode. The SyncNode is then passed as an argument to a static-function wrapper acting as the tick-hook function when a clock domain (or a reaction) is created. When a clock domain tick elapses, the actual tick-hook function pointed by the SyncNode is then activated. Tick-hook functions and synchronizing functions carry out the handshaking procedures. Finally, the data read by the wrapping module are passed to the libGALS program wrapped in the SYSTEMC module.

Figure 5.6 illustrates the chronological steps taken in synchronization between the libGALS program and the other SYSTEMC modules. Details of each step are described further in Table 5.1. The figure has been divided into two parts, the upper part representing activities carried out in libGALS program, and the lower governed by the SYSTEMC simulation kernel. Note that due to the single-thread simulation model of SYSTEMC, each module and the synchronizing function (which is in the libGALS-SYSTEMC module) take turns to be excuted. The libGALS program, which is running in other threads, is executed in parallel.

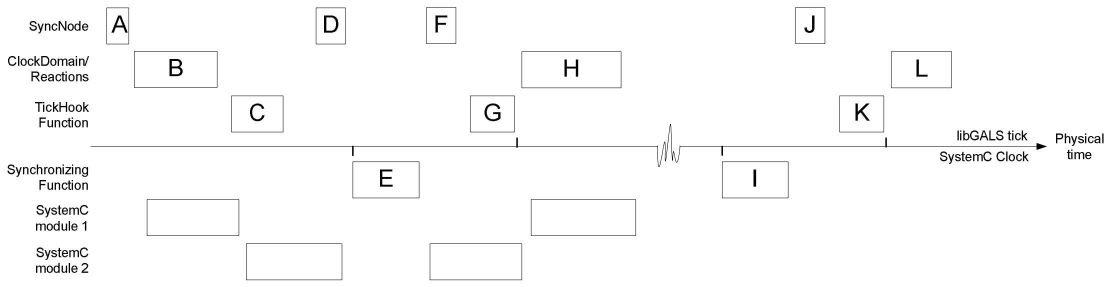


Figure 5.6: Timing diagram of libGALS-SYSTEMC synchronization

Table 5.1: Activites of libGALS-SYSTEMC synchronization

Stage	Description
A	Create SyncNode, initialize data structure of SyncNode, and register the tick function.
B	Clock domain reaches tick boundary (end of tick).
C	TickHook function is called by the libGALS. Outputs from the libGALS program have been generated to be used by other SYSTEMC modules. Time annotations of reactions are inserted here.
D	'signalSC' is called by the tick function to signal the signaling semaphore 1, that the tick-hook function is ready to synchronize with the synchronizing function. 'pendSC' is called to wait synchronizing function to reply.
E	Synchronizing function is activated by the clock signal from the SYSTEMC. Communications with other SYSTEMC modules are carried out.
F	'signallibGALS' is called by the synchronizing function to resume TickHook function. Inputs from other SYSTEMC modules are ready for the libGALS program. 'pendGALS' is called to await the next synchronization from the TickHook function.
G	TickHook function resumes, inputs to libGALS programs are registered. Start a new libGALS tick.
H	Clock domains/reactions start activities in the new tick (beginning of tick).
I	When SYSTEMC clock reaches the edge again. Refer to E.
J	Refer to F.
K	Refer to G.
L	Refer to H.

5.5 Programming model of GALS-Designer

A libGALS program is illustrated in Figure 5.7 to demonstrate how to integrate a libGALS program within the GALS-Designer. This example also shows that there is no need for extensive code modification between a libGALS model and GALS-Designer SYSTEMC modules. This enables automatic wrapping of the existing libGALS program into GALS-Designer modules.

The libGALS program is a composition of one or more asynchronous concurrent clock domains, illustrated as rounded rectangles (CD1 and CD2) in Figure 5.7. Each clock domain can include one or more synchronous concurrent behaviors/programs. These reactions are shown as rectangles (CD1_R1, CD1_R2, CD1_R3, and CD2_R1) within clock domains. To enable hierarchical design, each reaction can be further decomposed into child reactions. Such relationships are shown in Figure 5.7, where CD1_R1 and CD1_R2 are child reactions to CD1_R3.

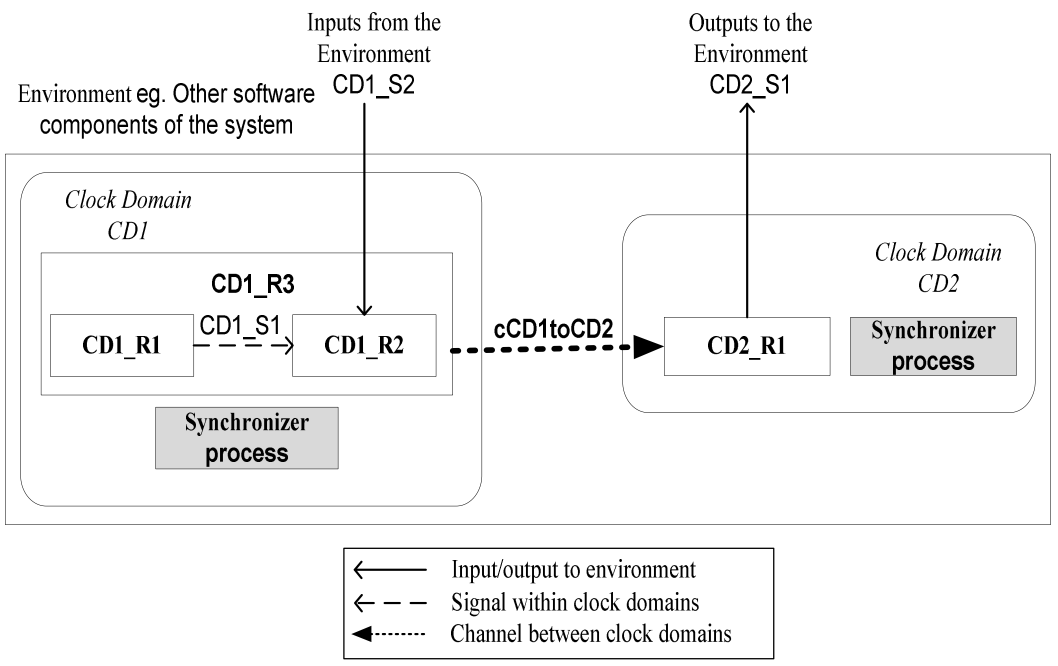

Inputs from the Environment
CD1_S2
Outputs to the Environment
CD2_S1
Environment eg. Other software components of the system
Clock Domain CD1
CD1_R3
CD1_R1
CD1_S1
CD1_R2
Synchronizer process
cCD1toCD2
Clock Domain CD2
CD2_R1
Synchronizer process
Input/output to environment
Signal within clock domains
Channel between clock domains

Figure 5.7: A libGALS program example

Communication between reactions of the same clock domain is via signals CD1_S1. Signals are also used for the interaction of reactions with the external environment to a libGALS program, e.g. CD1_S2 and CD2_S1. Reactions in different clock domains in the same libGALS program communicate through message passing over channels cCD1toCD2.

Listing 5.1 and Listing 5.2 are segments of a libGALS program which describes the GALS system illustrated in Figure 5.7. Listing 5.1 consists of the definitions of user-typed data (lines 2-5), clone function of the user-typed data (lines 6-11) and reaction functions (lines 12-65). User-typed data are used (1) in internal algorithms, (2) to define the value type of a signal and (3) to define the value type passed by a channel. In this example, data type of ‘customedType’ is used in the channel cCD1toCD2. Definitions of reactions represent the bodies of reactions which will be instantiated in clock domains. A definition of a reaction starts with the name of the reaction with the keyword REACTION_FUNCTION. Line 12 illustrates the starting point of defining

ReactionCD1R1. The body of a reaction is composed of the initialization and the behavior of the reaction. The initialization of a reaction starts with the API call ‘initReaction’ and ends with the API call ‘endInitReaction’ as shown on lines 13 and 17, respectively. Reactions are created with arguments and can be extracted by using ‘getArgument’ API call (lines 14 and 15). Channel is used to send and receive messages between reactions of different clock domains (lines 48 and 60, respectively). Signals emitted by a reaction are registered (line 16 and line 56). The code representing description of the behavior of a reaction is written after the ‘endInitReaction’ API call. Besides using any usual C sequential programming language constructs, reaction behavior can use a set of additional libGALS control statements to model flow control and reactivity in the form of API calls. The behavior of a reaction consists of arbitrarily mixed sequences of libGALS reactive and standard C statements (e.g. lines 18 to 22). Examples of libGALS control statements include:

1. ‘emit’ for broadcasting the presence of a signal, lines 19 and 62, to all reactions within the same clock domain.
2. ‘pause’ to explicate end of tick, as shown in lines 14 and 16.
3. ‘await’ to wait on the presence of a signal, lines 24 and 26.
4. ‘fork’ and ‘join’ to fork out and then wait for the joining of child reactions, lines 43 and 44. A parent reaction can proceed only if all of its forked child reactions have joined.

Details of available libGALS API calls can be found in Chapter 4. Reactions in different clock domains communicate through channels. A sending reaction has to prepare a message to send by creating the message (lines 45-47) followed by a ‘send’ (line 48) API call, which takes arguments including the instance of the channel, the message, and the type of the message. At the receiving side, a place-holder of the receiving message needs to be declared (line 59) prior to the ‘receive’ API call (line 60). ‘endReaction’ API call (lines 23, 34, 50, and 64) is used to denote the end of the behavioral description of reaction.

Listing 5.1: Definition of user-defined data types and reaction functions

```
#include “libGALS.h”
typedef struct customedType {
  // Definition of User Types
  int val;
} customedType;
customedType *customedType_clone(customedType* original) {
  customedType* newData =
      (customedType*)malloc(sizeof(customedType));
  newData->val = original->val;
  return newData;
}
REACTION_FUNCTION(ReactionCD1R1) {
  initReaction();
  signal CD1_S1 = (signal)getArgument(1);
  signal CD1_S2 = (signal)getArgument(2);
  registerEmitter(CD1_S1);
  endInitReaction();
  ...// Computational segments
  emit(CD1_S1, 0);
  pause();
  ... // Computational segments
  pause();
  endReaction();
}
REACTION_FUNCTION(ReactionCD1R2) {
  initReaction();
  signal CD1_S1 = (signal)getArgument(1);
  signal CD1_S2 = (signal)getArgument(2);
  endInitReaction();
  await(CD1_S1);
  ... // computational segments
  await(CD1_S2);
  ... // computational segments
  endReaction();
}
REACTION_FUNCTION(ReactionCD1R3) {
  initReaction();
  reaction CD1_R1 = (reaction)getArgument(1);
  reaction CD1_R2 = (reaction)getArgument(2);
  channel cCD1toCD2 = (channel)getArgument(3);
  endInitReaction();
  while(1) {
    fork(CD1_R1); fork(CD1_R2);
    join(CD1_R1); join(CD1_R2);
    customedType *dataToSend =
        (customedType *)malloc(sizeof(customedType));
    dataToSend->val = success;
    send(cCD1toCD2, dataToSend, customedType);
  }
  endReaction();
}
REACTION_FUNCTION(ReactionCD2R1) {
```

```
53      initReaction();
54      signal CD2_S1 = (signal)getArgument(1);
55      channel cCD1toCD2 = (channel)getArgument(2);
56      registerEmitter(CD2_S1);
57      endInitReaction();
58      while(1) {
59        customedType *dataToReceive;
60        receive(cCD1toCD2, dataToReceive, customedType);
61        if (dataToReceive->val == success)
62          emit(CD2_S1, 0);
63      }
64      endReaction();
65    }
```

Entities and objects of the libGALS programs, including clock domains, reactions, signals and channels, are created in Listing 5.2. Firstly, a libGALS program is created with ‘createlibGALSProgram’ call (line 4). Clock domains are instantiated by using ‘createClockDomain’ API call (lines 5-10). Channels, signals, and reactions have to be instantiated as arguments before being used to create other reactions. The channel cCD1toCD2 are created through the use of ‘createChannel’ API call (line 15), which takes the sending and receiving clock domains as arguments. Instantiation of signal objects is via ‘createSignal’ API call (lines 16-18). Reactions are then created with ‘createReaction’ API call (lines 19-55). Note that tick-hook function and its argument for both creations of clock domains and reactions are optional, that is, they can be substituted as 0 (or NULL) when calling the creation functions. Tick-hook functions can be used to synchronize with the other software components, such as SYSTEMC modules, as described initially in Section 5.3 and with more detailed description in the following sections. Clock domains are activated by using ‘startClockDomain’ API calls (line 56-57). Finally the libGALS program starts via calling ‘startlibGALSProgram’ in line 58.

Listing 5.2: libGALS program that creates CDs, channels, signals and reactions

```
1     #include “libGALS.h”
2     #include “ReactionFunctions.h”
3     int main(void) {
4       createlibGALSProgram();
5       clockdomain CD1 = createClockDomain(
6           InputC1,          // Input function to clock domain
7           OutputC1,         // Output function to clock domain
8           CD1TickHook,      // Tick-hook function, called every tick
9           CD1TickHookArgs); // Arguments to tick-hook function
```

```
    clockdomain CD2 = createClockDomain(
        InputC2,
        OutputC2,
        CD2TickHook,
        CD2TickHookArgs);
    channel cCD1toCD2 = createChannel(CD1, CD2, “cCD1toCD2”);
    signal CD1_S1 = createSignal(CD1);
    signal CD1_S2 = createSignal(CD1);
    signal CD2_S1 = createSignal(CD2);
    reaction CD1_R1 = createReaction(
        CD1,               // Clock domain that the reaction is in
        ReactionCD1R1,     // Reaction function
        0,                 // Active status
        CD1R1TickHook,     // Tick-hook function, called every tick
        TickHookArgs,      // Arguments to tick-hook function
        2,                 // Number of arguments to the reaction
        CD1_S1,            // First argument
        CD1_S2);           // Second argument
    reaction CD1_R2 = createReaction(
        CD1,
        ReactionCD1R2,
        0,
        0,
        0,
        2,
        CD1_S1,
        CD1_S2);
    reaction CD1_R3 = createReaction(
        CD1,
        ReactionCD1R3,
        1,
        0,
        0,
        3,
        CD1_R1,
        CD1_R2,
        cCD1toCD2);
    reaction CD2_R1 = createReaction(
        CD2,
        ReactionCD2R1,
        1,
        0,
        0,
        2,
        CD2_S1,
        cCD1toCD2);
    startClockDomain(CD1);
    startClockDomain(CD2);
    startlibGALSProgram();
  }
```

The forming of a libGALS-SYSTEMC module from wrapping a libGALS program is presented in Listing 5.3. This demonstrates a SYSTEMC description, which wraps up the libGALS program shown in Listing 5.2 into a libGALS-SYSTEMC module. Note that most of the original libGALS program, as from Listing 5.1, remains untouched, and it requires minimal effort to implement a libGALS-SYSTEMC module. A diagram representing this module is illustrated in Figure 5.8. The programming interface of libGALS and reactions declarations are included with the above mentioned ‘extern C { … }’ construct (lines 2-5, Listing 5.3). In line 6, a header file, libgals_sc.h, is included to provide macros and data structures which are parts of the libGALS-SYSTEMC compartments.

The transformation from an existing libGALS model to a GALS-Designer module is as follows. A SYSTEMC module named GALS_PROG is created (line 7) with a set of its member variables and functions (lines 9-27). Firstly, member variables representing a set of input and output signals are declared. Declared signals include the clock signals for each clock domain (line 9), and interfacing signals (lines 10 and 11). Member variables, such as clock domains, channels, signals, and reactions are also declared (lines 12-15). ‘SyncNodeParser’ (line 16) is the macro to create the static function wrapper. ‘SyncNodes’ are declared through ‘NewSyncNode’ macro (lines 17-19). Within the constructor of GALS_PROG (lines 26-47), SyncNodes are created through ‘createSyncNode’ (lines 29-31), providing arguments including the name of the libGALS-SYSTEMC module and the actual tick-hook function pointed by the SyncNode. ‘createlibGALSProgram’ (line 32) is still required to establish data structures to execute the libGALS components. Upon the creation of the clock domains and reactions, a ‘SyncNodeHook’ macro is used as the static function wrapper, providing the SyncNodes as the argument (lines 36, 41, and 49) which can be applied to both clock domains and reactions. The creations of other clock domains, channels, signals, and reactions are the same as in the original libGALS program. Activations of clock domains and libGALS programs (lines 56-58) are essential to enable the libGALS part of libGALS-SYSTEMC module to be up and running. Synchronizing functions are registered (lines 59-64) with the clock signals to the corresponding clock domains.

Examples of a tick-hook function for clock domain CD1, listed in lines 67-71, consist of handshaking operations and evaluations of interfacing signals. A corresponding synchronization function (lines 78-79) implements the counterparts of the handshaking to the tick-hook function. Note that in Listing 5.3, line 69, scCD1_S2 is a SYSTEMC signal and CD1_S2 is of type libGALS signal. The interfacing between the two kinds of signals is carried out within the hook function. Input signals to a SYSTEMC module are first checked and then emitted to the libGALS program (line 69). Similarly, output signals are written when they are present in the libGALS program (line 74). Synchronization between a tick-hook function and a synchronizing function is presented as the grey area in the Figure 5.8 and detailed in Section 5.3.

Listing 5.3: SYSTEMC module resulted from the libGALS program

```
1      #include "systemc.h"
2      extern "C" {
3        #include "syncapi.h"
4        #include "ReactiveFunction.h"
5      }
6      #include "libgals_sc.h"
7      SC_MODULE(GALS_PROG) {
8      public:
9        sc_in<bool> clk_CD1, clk_CD2;
10       sc_in<bool> scCD1_S2;
11       sc_out<bool> scCD2_S1;
12       clockdomain CD1, CD2;
13       channel cCD1toCD2;
14       signal CD1_S1, CD1_S2, CD2_S1;
15       reaction CD1_R1, CD1_R2, CD1_R3, CD2_R1;
16       SyncNodeParser(GALS_PROG);
17       NewSyncNode(GALS_PROG, snCD1);
18       NewSyncNode(GALS_PROG, snCD2);
19       NewSyncNode(GALS_PROG, snCD1_R1);
20       // Tick-hook functions
21       void CD1_TickHook();
22       void CD2_TickHook ();
23       void CD1_R1_TickHook();
24       // Synchronization functions
25       void CD1_Sync();
26       void CD2_Sync();
27       void CD1_R1_Sync();
28       SC_CTOR(GALS_PROG) {
29         snC1 = createSyncNode(GALS_PROG, CD1_TickHook);
30         snC2 = createSyncNode(GALS_PROG, CD2_TickHook);
31         snCD1_R1 = createSyncNode(GALS_PROG, CD1_R1_TickHook);
32         createlibGALSProgram();
```

```
    clockdomain CD1 = createClockDomain(
        InputC1,
        OutputC1,
        SyncNodeHook(GALS_PROG),
        snCD1);
    clockdomain CD2 = createClockDomain(
        InputC2,
        OutputC2,
        SyncNodeHook(GALS_PROG),
        snCD2);
    // The same as lines 15 to 19 in Listing 5.2
    // to create channels and signals
    reaction CD1_R1 = createReaction(
        CD1,
        ReactionCD1R1,
        0,
        SyncNodeHook(GALS_PROG),
        snCD1_R1,
        2,
        CD1_S1,
        CD1_S2);
    // The same as lines 28 to 55 in Listing 5.2
    // to create reactions
    startClockDomain(CD1);
    startClockDomain(CD2);
    startlibGALSProgram();
    SC_METHOD(CD1_Sync);
    sensitive << clk_CD1.pos();
    SC_METHOD(CD2_Sync);
    sensitive << clk_CD2.pos();
    SC_METHOD(CD1_R1_Sync);
    sensitive << clk_CD1.pos();
  }
};
void GALS_PROG::CD1_TickHook() {
  snCD1->signalSC();
  if(scCD1_S2.read()) emit(CD1_S2);
  snCD1->pendlibGALS();
}
void GALS_PROG::CD2_TickHook() {
  snCD2->signalSC();
  scCD2_S1.write(present(CD2_S1));
  snCD2->pendlibGALS();
}
void GALS_PROG::CD1_R1_TickHook() { ... }
void GALS_PROG::CD1_Sync(void) {
  snCD1->pendSC(); snCD1->signallibGALS();
}
void GALS_PROG::CD2_Sync(void) { ... }
void GALS_PROG::CD1_R1_Sync(void) { ... }
```

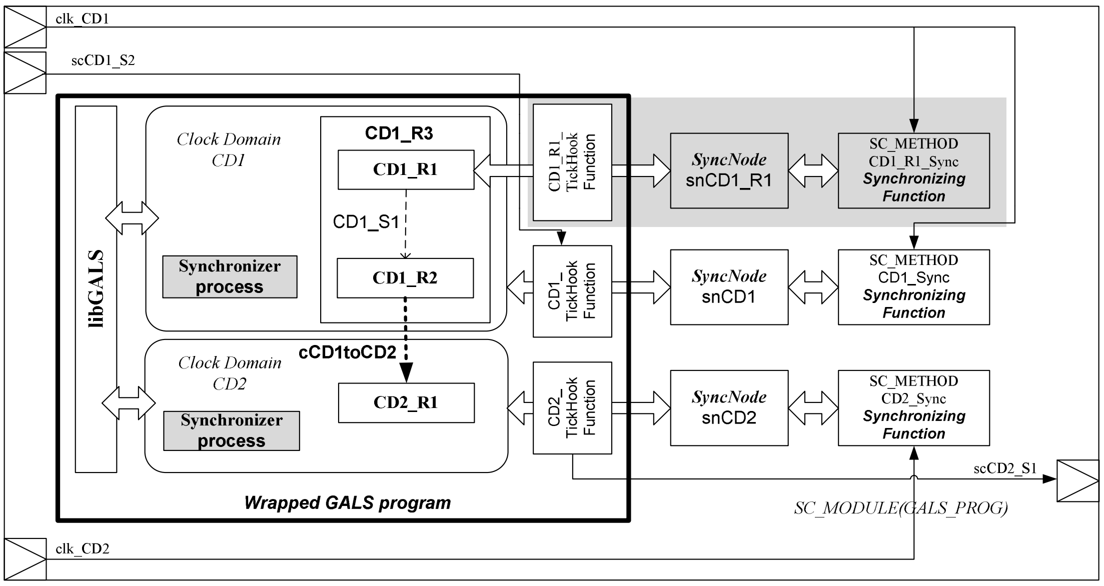
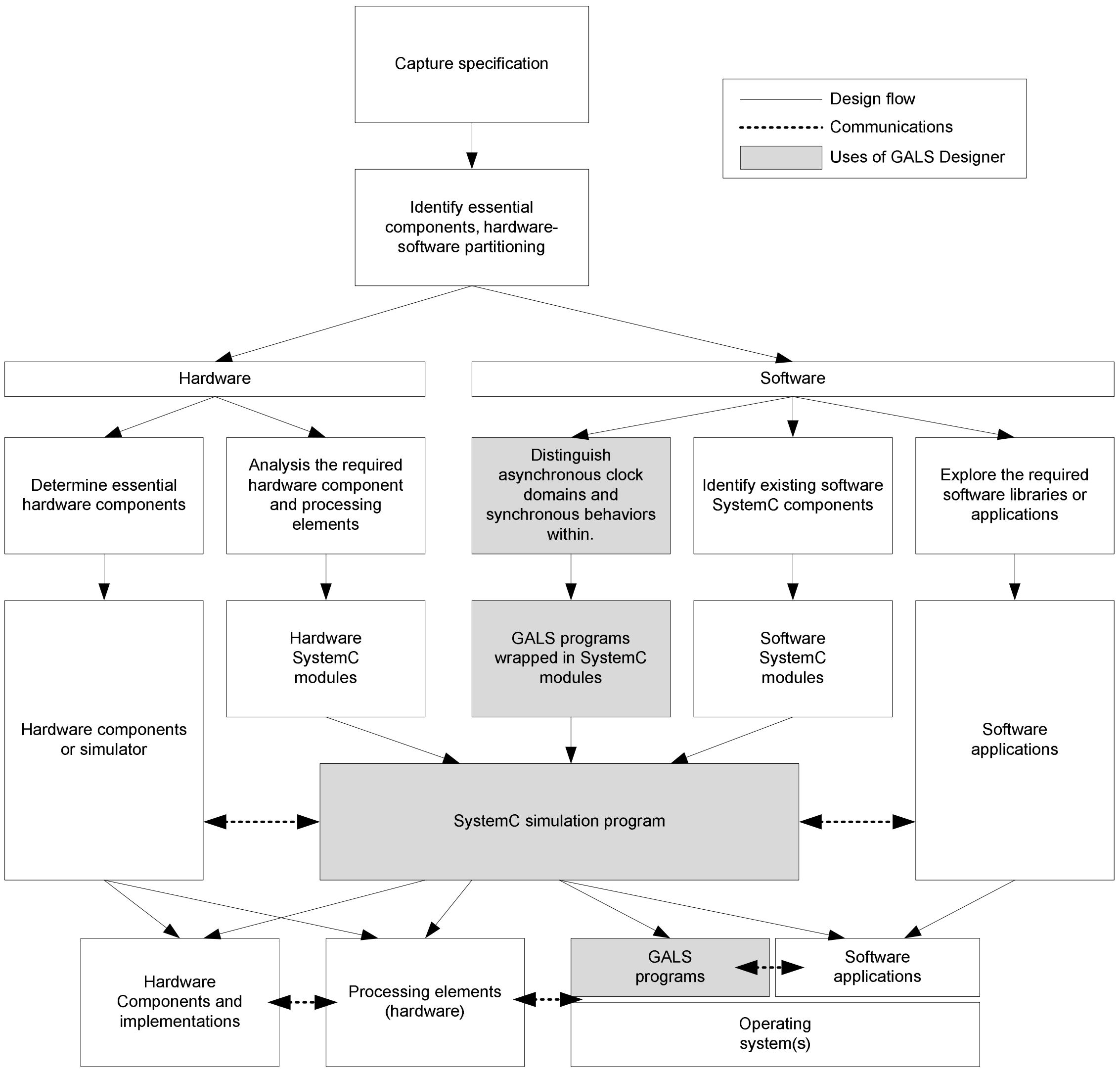


Figure 5.8: Integration of libGALS program into a SYSTEMC module

In the libGALS-SYSTEMC module, reactions are still executed in the same fashion as in the libGALS program. However, contrary to the conventional simulation of a SYSTEMC executable which is single-threaded, the executables consisting of libGALS-SYSTEMC modules are multithreaded and can take advantage of being executed on multicore systems. Tick-hook function also enables the modeling of further details of communication and synchronization between the libGALS programs and the SYSTEMC wrapping module. For example, timing annotations can be inserted into tick-hook functions and then used for architecture exploration and performance evaluation, as detailed in Section 5.7.

5.6 GALS system design using GALS-Designer

Figure 5.9 illustrates design flow in which the GALS-Designer is used. Solid lines represent the flow between design stages. Dashed lines represent the communications between components. GALS-Designer is utilized in stages shaded in grey. After system specification capture, hardware/software partitioning is performed. Software and hardware components can be categorized into two groups: existing components or those needed to implement. libGALS programs, which are derived from identified asynchronous and synchronous behaviors, are wrapped to become SYSTEMC modules that are integrated to a SYSTEMC simulation model of the designed system, which communicates with hardware/software simulators and performs the entire system simulation. As refinements of the design are carried out along with simulations and validations, the results lead to the final implementation of the system. libGALS programs and other software applications are executed with the support of operating systems on the same designated platform.


Capture specification
Design flow
Communications
Uses of GALS Designer
Identify essential components, hardware-software partitioning
Hardware
Software
Determine essential hardware components
Analysis the required hardware component and processing elements
Distinguish asynchronous clock domains and synchronous behaviors within.
Identify existing software SystemC components
Explore the required software libraries or applications
Hardware components or simulator
Hardware SystemC modules
GALS programs wrapped in SystemC modules
Software SystemC modules
Software applications
SystemC simulation program
Hardware Components and implementations
Processing elements (hardware)
GALS programs
Software applications
Operating system(s)
Figure 5.9: GALS-Designer in system development

libGALS requires standard features provided by the operating systems, in that it guarantees the same behavior and outputs regardless as to which operating system is used. This simplifies the SYSTEMC simulation model, which does not require a target OS model. However, the target OS model can be included for fine-grained simulation. For example, when OS API calls are made by the libGALS, the required information can be passed to the OS model through the synchronizing function.

The concept of the libGALS enables the developer to describe GALS systems in a simple manner, without putting effort into how actual communication and synchronization between reactions and clock domains are carried out. These details are hidden by using libGALS, which guarantees the compliance of the designed system with the GALS MoC. Because the libGALS library is written in C, it is highly portable and has been ported to a range of operating systems, from non-real-time to real-time, such as Linux, Windows, uCOS-II, FreeRTOS, eCOS, and RTEMS, as detailed in Chapter 4. On the other hand, SYSTEMC allows modeling at different levels of abstraction, which makes it suitable as a development framework, demonstrated by many previous research and development efforts. SYSTEMC also enables designing systems using either top-down (system-level design) or bottom-up (component-based design) approaches according to the specific requirements of the applications [Cai & Gajski, 2003]. Both libGALS and SYSTEMC can be used to describe a system in different design phases that include: (1) specification, (2) modeling and analysis, and (3) implementation phase. The GALS-Designer development framework, which supports the design process in different design phases, is illustrated in Figure 5.10.

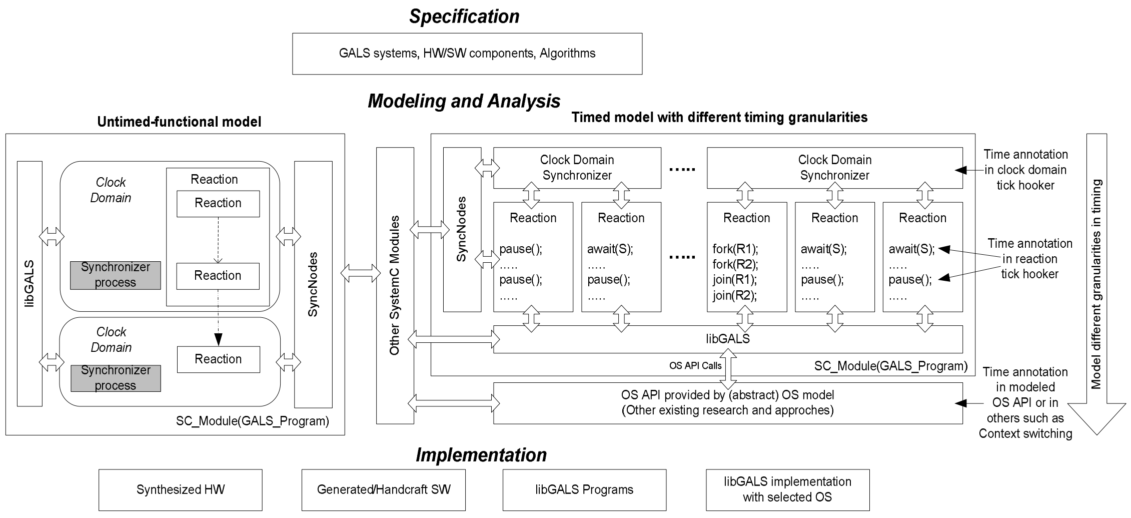


Figure 5.10: Development framework of the libGALS-SYSTEMC model

In the specification phase, libGALS is used to identify essential clock domains and concurrent reactions of a GALS system. Reactions within a clock domain do not need to be modeled with details of its actual implementations; i.e. clock domains can contain a single reaction (which can be refined into multiple reactions later) and such libGALS program can be referred to as a 'simplified libGALS program'. In this phase simplified libGALS programs are wrapped into SYSTEMC modules as described in the previous section. An overall system can consist of one or more libGALS programs and other components (hardware descriptions or software modules). Other system components, which do not follow the GALS MoC, are specified using SYSTEMC or other specification methodologies that can be incorporated within SYSTEMC.

At the next modeling and analysis phase, descriptions of SYSTEMC components are further refined into more concrete models of hardware and software. Models of these components can be at different levels of abstraction depending on what intellectual property (IP) vendors and designers have provided. Simplified libGALS programs are refined with more synchronous reactions, where reactions are described in further detail including:

1. Identification of concurrent behaviors within a clock domain that are modeled as separate reactions.
2. Introduction of the algorithms that perform data transformations in each of these reactions.
3. Specification of control and dependencies between reactions that are achieved via signal emit/await and fork/join API calls.

The number of clock domains that libGALS can support is practically unlimited (assuming the memory to store clock domain data structure is sufficient), and are bound by the underlying OS features. Grouping of clock domains into different libGALS-SYSTEMC modules (i.e. libGALS programs) is the designer's decision and is illustrated in Figure 5.11.

Each libGALS-SYSTEMC module represents a possible mapping to a separate processor or a libGALS program running on the target OS. This approach enables the implementation of heterogeneous systems, which include different processors with different computing power and therefore can execute clock domains of different complexities and different speeds. An example is shown in Figure 5.11 (a), where all clock domains are modeled and implemented on a single processor, as clock domains communicate with other SYSTEMC modules through necessary mechanisms. If a faster execution speed is required, clock domains can be mapped to separate processors as Figure 5.11 (b).

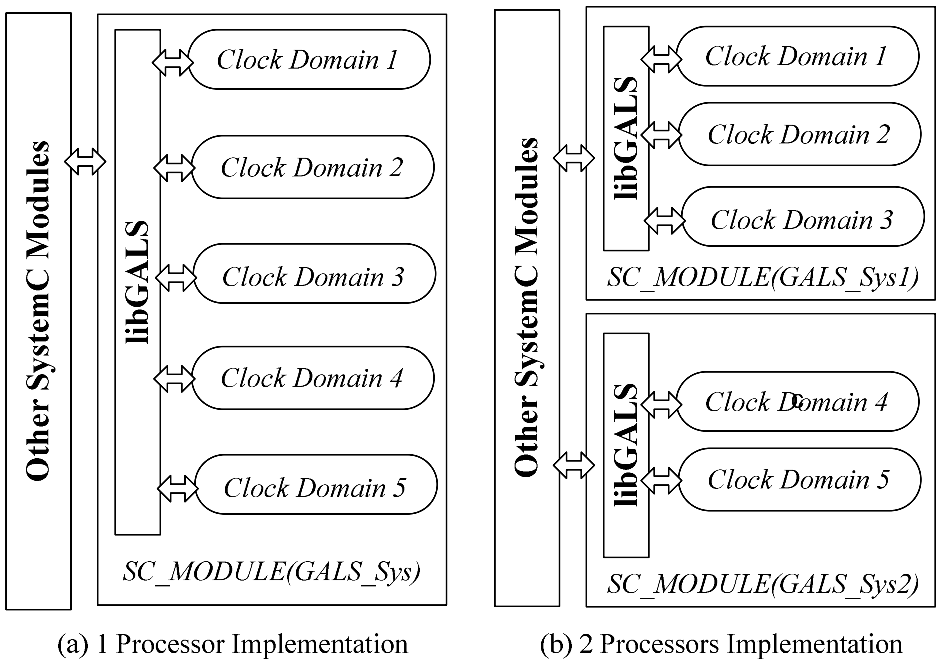


Figure 5.11: Clock domains mapped to different libGALS-SYSTEMC modules

Models of libGALS-SYSTEMC modules can be described as untimed or with different timing granularities by annotating timing for accurate simulation. Timing annotations can be made at clock domain level, reaction level, and operating system level. Execution times can be obtained, for example, through profiling and using instruction set simulators (ISS). At the clock-domain level, times are annotated within the tick-hook functions of the clock domains. This gives the designer information as to how the clock domains perform on different configurations of processors, enabling architecture exploration. To obtain higher accuracy, timing information can be further inserted to tick-hook functions of reactions. Other works have modeled and described abstract OSs which provide APIs that can be used by the application models and enable timing analysis as in [Posadas et al., 2005]. Similarly, libGALS is implemented by using common (RT)OS services whose models are already available. Because timing information can be annotated when simulating with libGALS and abstract OS APIs, more accurate simulations are possible. Modeling with different timing granularities enables trade-offs between the simulation performance and accuracy. As one extreme, an ISS can be used to execute libGALS programs to obtain the most accurate execution time, but with the slowest simulation speed.

Finally, at the implementation level, SYSTEMC modules are mapped to synthesized hardware or software generated automatically or manually as presented in [Cesario et al., 2002] and [Posadas et al., 2005]. libGALS-SYSTEMC modules are mapped (translated by a text parser) to libGALS programs for specific selected operating system used on the target processor(s).

5.7 Case studies and results of using GALS-Designer

To demonstrate the use of the GALS-Designer approach and how libGALS-SYSTEMC modules can be integrated with other SYSTEMC modeled components, an Internet-enabled frequency relay (IEFR) has been used, as illustrated in Figure 5.12. A similar model without network support [Radojevic et al., 2006] has presented the major components of the frequency relay in SYSTEMC. In Chapter 4, the libGALS model of the frequency relay was introduced. The frequency relay measures frequency in the electrical power system and the rate of its change, and switches on and off the loads in order to help maintain overall system frequency within the specified range. The IEFR is formed by coupling a frequency relay with a simple web server. IEFR enables communication with a Web Browser via the Internet to configure settings of the frequency relay, as well as to display status of its operation.

Clock domain and reaction partitioning are based on the characteristics of the relay. Four clock domains have been identified: data sampling, relay control, web service, and status gathering. Clock domains can be instantiated in different libGALS programs because of the requirements of the system or the capability of the execution platform. For example, a platform might not be powerful enough to host all four clock domains because the data sampling and relay control have high computational demand. To demonstrate that clock domains can be further allocated to different libGALS programs, data sampling and relay control are grouped in one libGALS-SYSTEMC module, as an example of the design decision. The other module contains the remaining IEFR functionalities. Note that the allocation of clock domains to the libGALS programs is driven by the characteristics of the application and based on the design analysis.

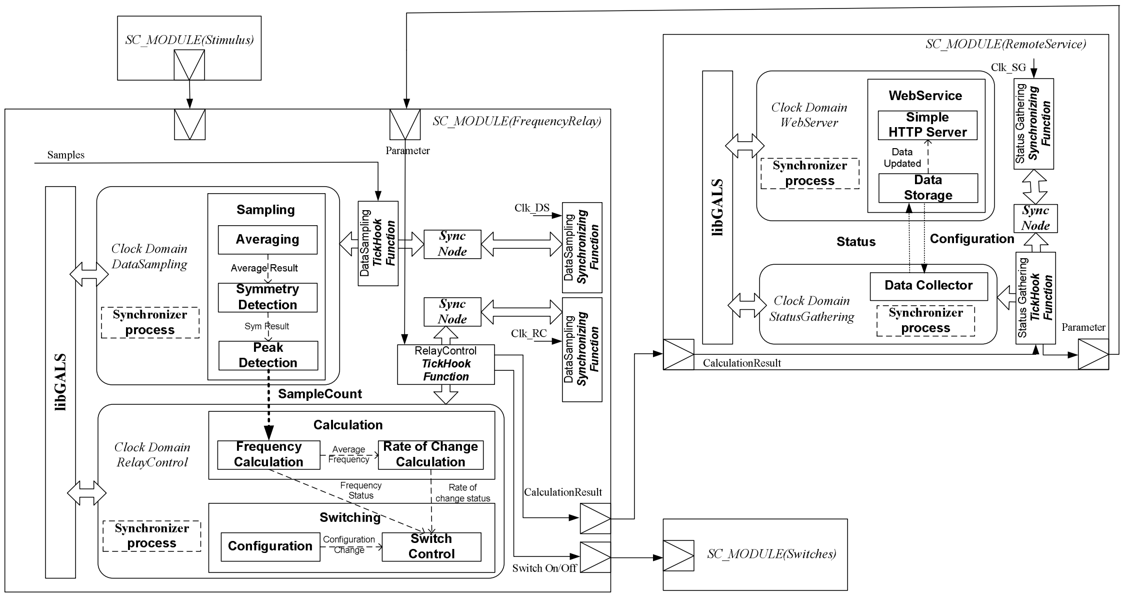


Figure 5.12: Internet-enabled frequency relay modeled with libGALS-SYSTEMC

Communication between clock domains ‘DataSampling’ and ‘RelayControl’ in module ‘FrequencyRelay’ are via channel ‘SampleCount’. Similarly, ‘WebServer’ and ‘StatusGathering’ of module ‘RemoteService’ exchange information through channels ‘Status’ and ‘Configuration’. Inter-module clock domains communicate with each other through SYSTEMC signals or channels, named ‘CalculationResult’ and ‘Parameter’ which can be modeled as the environment to the corresponding clock domains, or can be described as libGALS channels if GALS MoC is required. To simulate the overall system, inputs and outputs are provided and collected by SYSTEMC modules. Input stimulus, which is the digitized electric power signal waveform, is described in the module named ‘Stimulus’. Outputs, which are signals controlling the switches, are modeled in the SYSTEMC module called ‘Switches’. Interconnection between SYSTEMC modules is achieved through SYSTEMC signals. The simulation model of IEFR is performed on the Linux, where two libGALS programs communicate with each other through SYSTEMC channels. The corresponding implementation of such a model uses inter-process communication (IPC) of the host operating system.

Standard SYSTEMC executable is a single-threaded program, which cannot take advantage of using the state of the art multiprocessor platforms that are readily available. On the other hand, the libGALS-SYSTEMC model can take advantage of multiple processors or cores. The simulation speed can be increased and this can be demonstrated by simulating the libGALS-SYSTEMC models with a different number of processor cores. The results of simulation of the FrequencyRelay module from the IEFR, along with a number of other examples, are shown in Figure 5.13. The name of the example also indicates the number of clock domains in the model, for instance, ‘2CD FreqRelay’ represents a FrequencyRelay modeled with two clock domains.

Data Comp examples are synthetic examples, which consist of one or more clock domains as indicated by their names. Each clock domain consists of two reactions, one performing heavy computation within each tick and the other having the communication function of sending out results to the other clock domains through channels. They are designed in such a way as to present the performances of heavy data-driven computations with low data dependencies between clock domains.

Such examples are typical for video encoding and decoding applications, which include both audio and video parts. ‘3CD Kite Controller’, detailed in Chapter 4, consists of three clock domains that have a mix of data computations and control found in typical heterogeneous embedded systems. ‘2CD AsyncProto’ [Lavagno & Sentovich, 1999] is described by two clock domains. Experimental runs were carried out on an Intel Core 2 Quad 2.4GHz with 4GB of RAM with Linux 2.6.29.6 as the host OS. A different number of cores are set and made available to the OS by providing maxcpus=n, n = 1-4**,** as the argument to Linux kernel during the boot process. Average tick times (in µs) for all clock domains are obtained by running all programs for at least 10 million ticks. Simulations of pure SYSTEMC models, in which the same functionalities would be achieved without libGALS, are not carried out because of the following:

1. Noticeable modeling effort is required to implement the GALS MoC in SYSTEMC, since one might eventually implement functionality close to libGALS.
2. SYSTEMC does not provide certain control statements such as explicit pre-emption construct, and libGALS does. Models that use pre-emption statements would lose the abstraction intended by libGALS.
3. SYSTEMC kernel does not support simulations by employing by multicore simulation hosts.

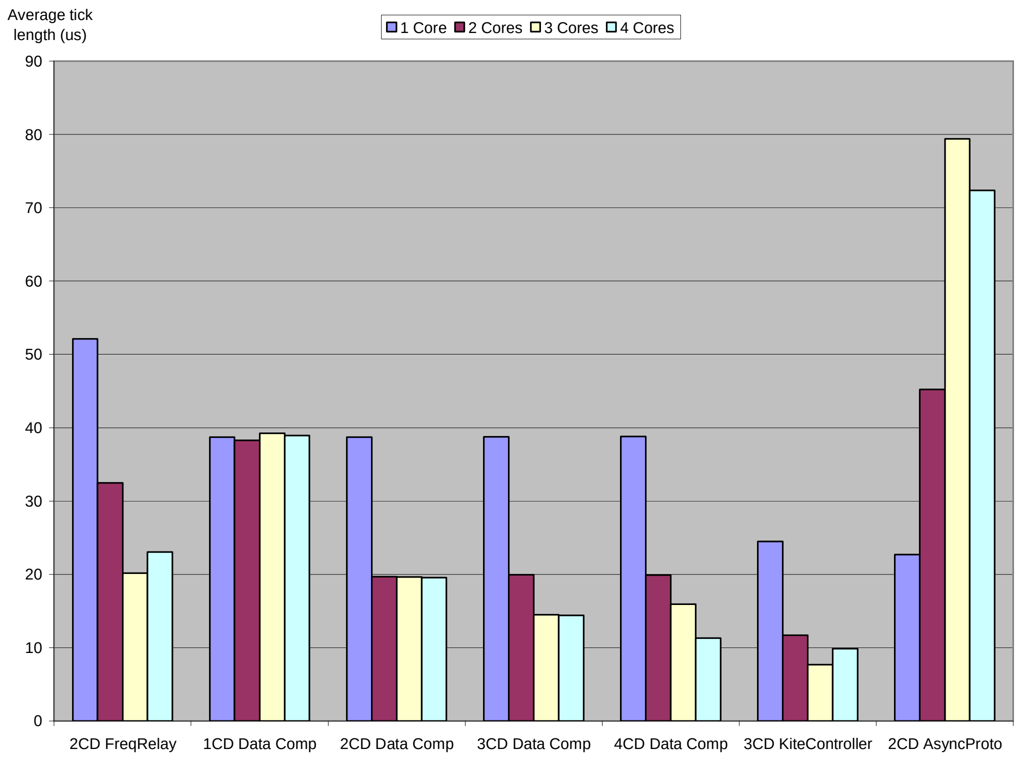


Figure 5.13: Simulation execution results of libGALS-SYSTEMC models

The simulation runs have shown that, in general, libGALS-SYSTEMC models perform faster when using more cores, with performance increasing as the number of clock domains and cores increases. Computations in ‘Data Comp’ are with low data dependencies and make use of parallelism to demonstrate the advantage of running on the multicore systems. 2CD FreqRelay, 3CD Kite Controller, and 2CD AsyncProto when executed on four cores, do not achieve performance gain as would be expected, because reactions of the same clock domain are distributed on different processors, which, in turn, results in overheads of synchronizations and program migrations between processors. Such a situation appears more obviously if two clock domains are highly dependent on each other (with frequent exchange of data), as is the case in 2CD AsyncProto example. That is, one clock domain is the sender and the other is the receiver. Both sender and receiver are blocked when waiting for the rendezvous in channel communication. The blocking-releasing order of both clock domain executions will result in only one thread running at a time while the other thread from the other clock domain is waiting for the communication to occur. This leads to the sequentialization of the activities of communicating clock domains and reduces the benefit of the multicore platform.

5.8 Summary

In this chapter, a new design framework, GALS-Designer, for the design of complex GALS software models in C programming language using libGALS library, as well as their integration with other components described in SYSTEMC, is introduced. libGALS models wrapped into SYSTEMC modules, called libGALS-SYSTEMC modules, are capable of communication with other SYSTEMC modules. libGALS-SYSTEMC modules can use different levels of abstraction in different design phases and with different timing granularities. Taking advantage of the libGALS multithreaded implementation, such modules can execute on multiprocessor and multicore platforms, opposite to standard SYSTEMC models which are single threaded.

Furthermore, as libGALS has been ported to a number of OSs, as detailed in Chapter 4, the same libGALS program, with practically no modifications, can be used in the simulation on one (host) and can be later implemented on the target OS with minimal efforts. This demonstrated the use of the approach on a complex embedded systems design. As a case study, the model of Internet-enabled frequency relay was first constructed and was then implemented as a libGALS program. Finally, the simulation performance of a number of examples has been analyzed when using a computer with different numbers of cores. It was shown the libGALS-SYSTEMC approach can take advantage of those cores, which is not possible when using standard SYSTEMC.

6

Dynamic system designs in DynamicGALS

This chapter presents the DynamicGALS framework, which enables the design of Dynamic Globally Asynchronous Locally Synchronous (DGALS) systems in the C programming language. A DGALS system consists of multiple DGALS programs and can be executed on platforms ranging from a single-processor to multicore and distributed systems. A DGALS program itself consists of a variable number of concurrent asynchronous behaviors at the top level of program hierarchy, which run on a single or multicore computational node. Each asynchronous process can be naturally composed of a number of synchronous concurrent processes. The mechanism for creation, termination, and mobility of asynchronous behaviors allows any existing behavior to create other asynchronous behaviors in their own or any other DGALS programs, regardless of their location. In this way, the overall system adapts to changes in the environment and the execution platform dynamically.

The DynamicGALS framework consists of a library named libDGALS, which also provides a run-time support for execution of DGALS programs. Features of libDGALS are available in the form of application programming interface (API) to the software designers. libDGALS, which is an extension of libGALS and can be built on top of almost any operating system, is highly portable and has low run-time memory requirements. In contrast to the GALS-Designer approach in Chapter 5 that systems of multiple GALS programs are modeled statically, DGALS programs in DynamicGALS framework are instantiated dynamically.

6.1 The need for framework to design dynamic systems

An increasing number of computing applications connect the computing world with the physical world, creating a single system, often called a cyber-physical system (CPS) [Krogh et al., 2008]. Most CPSs have some common features: (1) a distributed execution environment with computation nodes and their interfaces with the physical world connecting or disconnecting from the system at any time, (2) system functions are implemented as concurrent behaviors that may be synchronous or asynchronous each to the other, and (3) functions and behaviors have a lifetime and can be created and terminated dynamically. The goal is to allow the execution of such systems with high autonomy and cater for dynamic changes in both the physical world and the execution platform itself. Such CPSs need a high degree of run-time adaptivity, to enable them to survive situations such as a loss (or addition) of a computation node; loss (or addition) of interfaces to the physical world; variations in frequency and nature of requests for computation on any node; the ability to react in time on important events regardless of the current system load; etc. An example of such a CPS is a security surveillance and access-control system installed over large areas like cities, airports, commercial centers, etc, consisting of a huge number of disparate sensors connected with computers into sensor nodes, each capturing information in real-time and collaborating to achieve the final goal of object tracking and threat detection.

Such a complex CPS is difficult to design and implement because of the concurrent and asynchronous execution of various sensor nodes, synchronization and transfer of data between the nodes, fault tolerance and recovery, and finally the utilization of heterogeneous execution and communication architectures (e.g. combination of distributed and shared memory) as the execution platform. Obviously, such systems have a high degree of inherent non-determinism, so controlling this non-determinism and providing a consistent behavior in different scenarios would indeed be an ideal goal. Yet, this is difficult to achieve with current programming languages and practice.

On the one hand, sequential programming languages, such as C and C++, which are most often used in the implementation of current CPSs, lack the ability to program basic safe concurrent behaviors with the proper level of determinism and reactivity to the events from the physical world. Applying a formal Model of Computation (MoC) to CPS designs allows one to validate and even possibly verify the correctness of the critical components of these systems. A correctly chosen formal MoC also allows the designing of a complex system by composing simpler parts. For instance, the GALS [Chapiro, 1984] MoC, which describes concurrent asynchronous and synchronous behaviors, lends itself well to a significant number of complex CPSs. ‘Asynchronous concurrency’ is suitable for programming behaviors that run at their own pace, controlling their respective sensors, and communicating occasionally. ‘Synchronous concurrency’ might be a better choice for programming concurrent behaviors that are running on a single computation node to reduce overheads, as they communicate more frequently with each other, and at the same time guarantee key system properties such as deterministic behavior.

However, the GALS MoC lacks the ability to describe the dynamic nature of the majority of CPSs, such as creating behaviors at the other computational node at run-time. This leads towards evolving the GALS MoC from the static to the dynamic case, called ‘Dynamic GALS’ or ‘DGALS’. A framework approach is needed for both the design of CPSs and run-time support for dynamics of the CPSs by honoring the DGALS MoC. The DynamicGALS framework is proposed for such needs and is detailed in the following sections.

Related works and approaches are presented in Section 6.2. In Section 6.3 an abstract design is used as an example to underpin the principles behind the DynamicGALS framework and its features. A more complex example of a DGALS system which is both dynamic and distributed is demonstrated in Section 6.4. Section 6.5 presents the internal implementation details of the DynamicGALS framework, while

Section 6.6 then provides the benchmarking results when libDGALS was used in a number of other applications. Finally, a summary of this chapter is provided in Section 6.7.

6.2 Related works and the DGALS approach

Adequate frameworks provide a means for designing systems, and support the execution of deployed systems. Libraries that provide programming interfaces, and languages that provide essential constructs, are used to describe systems under design. To support the deployment and execution of both prototyped and final implementations of these systems, run-time environments are essential. Frameworks targeted at the design of complex systems need to meet a number of requirements to be effectively used by system designers. What follows is a comprehensive, but in no way exclusive list of the requirements that need to be satisfied by any framework that supports programming complex dynamic systems:

1. *Behavior and internal encapsulation*: The programming framework should allow the decomposition of the system into smaller manageable behaviors and the easy composition of these behaviors into an overall system. Also, the framework needs to support static (at design time) and dynamic (at run-time) instantiation of these concurrent behaviors.
2. *Safe communication*: Concurrent behaviors need to communicate. Safe mechanism for synchronization and communication between concurrent entities should be a primitive construct in the framework. Communication between concurrent entities should hide the details of the underlying communication layer, i.e., some concurrent entities in the system might be running in a distributed memory environment, while others might be running in a shared memory environment, but the higher-level programming abstractions used should be the same.
3. *Location transparency and mobility*: The designer should have no need to change the designed system behaviors, when the underlying infrastructure changes, or the required changes should be at least minimal. This is known

as location transparency. The ability of behaviors to move from one physical location (computational node) to another is essential in dynamic systems. For example, some piece of code not available on a computation node might be obtained from a code repository at run-time and activated as needed.

4. *Fault tolerance and possible recovery*: A large complex dynamic system is bound to have failures. Any design framework geared towards such systems needs to provide built-in, error-tolerance capabilities and possibly recovery.
5. *Automated formal validation and possible verification*: The design of complex systems needs to be approached from a system-level design perspective rather than a programming perspective. The framework should support a formal MoC, which, as mentioned previously, allows system designers to formally validate and possibly verify certain critical aspects of the designed systems.
6. *Reactivity and abstract data fusion*: Every incoming event to the designed systems needs to be responded to. Programming such 'reactive' [Harel & Pneuli, 1985] behaviors can be made easy by providing programming paradigms especially suited for data fusion from multiple sensors or other sources.
7. *Ability to take advantage of the heterogeneous execution and communication platforms*: The physical infrastructure (i.e. targeted processor architectures, or computing platforms) that the software system are executed on might consist of a heterogeneous set of computational elements, each element can be implemented by using single-processor and multicore CPUs and GPUs. Even the communication layer (i.e. adaptors and buses) is to be heterogeneous. The underlying physical infrastructure and the designed system behaviors should be separate, and the framework should allow the change of one, without affecting the other. This improves the overall reliability, portability, and flexibility of the designed system.

8. *Ability to accommodate legacy code*: There are large software applications, which have been written in traditional programming languages like C/C++, so any new programming framework should be able to accommodate and interface with these software applications with minimal or no changes at all.

Not many programming languages and frameworks excel in all the aforementioned requirements. Traditional programming languages like C, C++, and Java lack either the basic mechanisms to describe concurrency and/or safe communication between behaviors implemented using threads [Lee, 2006]. Recently, a number of programming frameworks and languages that target dynamic system development have been proposed. All these have advantages and drawbacks.

Integrating asynchronous concurrent behaviors into bigger systems is also known in the world of ‘actors’ [Hewitt et al., 1973][Clinger, 1981], where asynchronous actors communicate with each other using message-passing mechanisms. There are a number of implementations in the form of libraries or frameworks added to existing programming languages such as Actor Foundry [Astley, 1999], Scala Actors [Haller & Odersky, 2009] (both implemented using Java and running on JVM), or included into new concurrent languages Erlang [Armstrong et al., 1993]. However, message passing between actors is sometimes implemented as passing-by-reference (rather than creating a deep copy of the object to pass), which violates the semantics of the Actor model. Passsing-by-reference will not work in distributed-memory architecture because referencing to memory at a remote site is not possible. Also, the Actor-based systems provide a general asynchronous model, which is essential for majority of clustered distributed dynamic systems. However the Actor model does not allow explicit grouping of actors or internal concurrent behaviors within an actor that would perform synchronously. Finally, and most importantly, they lack the ability to react to events in the environment. A similar case can be made for multi-agent systems, such as JADE [Bellifemine et al., 2005], which provide for reactivity, but at the expense of huge execution overhead (i.e., large run-time library) on computation nodes.

There are approaches to create languages to implement formal MoC to describe concurrency and communication between asynchronous behaviors, as the remedies to general thread programming. For instance Occam [Galletly, 1990] implements the CSP [Hoare, 1978] MoC. However, both of the above mentioned approaches lack a support for mobility of behaviors. Extensions to support mobility have been made to Occam, resulting in Occam-pi [Welch & Barnes, 2005]. However, these languages lack the constructs to describe complex data structures and algorithms. To resolve this problem, as an example, CSP has been implemented in software libraries of general programming languages, such as JCSP [Welch et al., 2002], CTJ [Hilderink et al., 1999], and Scala [Odersky et al., 2004] (on the top of the Actor-based model) in Java, and CCSP [Moores, 1999] in C, but mobility is not supported in these languages. [Barnes, 2005] presents a technique to interfacing both Occam-pi and C, to obtain both mobility and support for data-driven computations. However, it complicates the design process without having a single-language environment.

Some attempts with the tools and frameworks are centered on the concepts of distributed systems, such as X10 [Charles et al., 2005]. In X10, asynchronous behaviors are called ‘activities’ running on distributed ‘places’. However, X10 is not based on a formal MoC. Other languages, such as Axum [Microsoft Corporation, 2008], take into account current languages and legacy codes, but also rely on powerful and heavy virtual machines (the .NET framework), which abstract away the underlying platform to enforce heterogeneity of the execution environment.

Languages and platforms which emerge from the synchronous/reactive MoC [Benveniste & Berry, 1991][Boussinot, 1996] and the mobile agent-based approach [Fuggetta et al., 1998] also exist, such as RAMA [Nikaein, 1999], and REJO [Acosta-Bermejo, 1999] along with its platform ROS [Acosta-Bermejo, 2000]. They provide mobility and reactivity, but not one provides constructs for asynchrony of behaviors and communication between behaviors, which is required and natural in the distributed systems. They also lack the features for communication and interaction of groups of synchronous agents.

One example of a systematic approach, which merges synchrony with asynchrony in a formal GALS model, to the design of complex static systems, is shown in [Gruian et al., 2006] and [Malik et al., 2010] where the language called 'SystemJ' was introduced. Such an approach contributes to fast and reliable design of software systems. Yet the SystemJ approach suffers from a number of limitations: (1) concurrent asynchronous and synchronous behaviors, called clock domains and reactions in SystemJ programs, respectively, are compiled to sequential and static codes; i.e., a designer cannot instantiate new clock domains at run-time. Therefore one cannot design dynamic systems. (2) SystemJ, which extends the Java language and uses the Java Virtual Machine (JVM), is far too abstracted from the underlying platform to properly utilize heterogeneous execution architectures. For example, a designer is unable to assign processor affinities to the clock domains, thus leaving this as the decision of the underlying JVM and the operating system. Accessing hardware features still requires programming in different host languages to cooperate with the JVMs. (3) Finally, SystemJ does not provide a suitable and efficient mapping on multicore execution targets and does not provide inherent support for programming distributed architectures (e.g., networked systems).

An extension of SystemJ, called Dynamic SystemJ (DSystemJ), which supports DGALS MoC, has been recently proposed [Malik et al., 2010]. It extends SystemJ with behavior creation and termination mechanisms and weak mobility (behavior migrations without state capture), but still inherits the dependency on the JVM.

Other approaches such as MPI [Gropp et al., 1999] and OpenMP [Dagum & Menon, 2002] are based on the use of C/C++, but are limited to static systems (MPI-1 and OpenMP), or to dynamic systems, but lacking process mobility (MPI-2) and reactivity. Finally, both these approaches (MPI and OpenMP) lack an all-encompassing formal MoC.

Almost all of the above mentioned approaches, except SystemJ and DSystemJ, are based on a single level of concurrency in the form of either asynchrony (such as processes in CSP, and Actor model) or synchrony (e.g. RAMA and REJO/ROS); some of them do not follow any formalism (e.g. OpenMP and MPI).

The formal DGALS MoC, extended and benefited from GALS MoC, covers the required features to program complex real-world dynamic systems. The DynamicGALS framework based on the DGALS MoC, provides libDGALS, a library for programming DGALS systems, as well as run-time support. libDGALS builds on the libGALS library introduced in Section 4 used for designing static GALS systems. While preserving features of libGALS with minor modifications, libDGALS significantly enhances the power and applicability of the design framework.

6.3 Overview of the DynamicGALS framework

From the discussion in the previous section, the DynamicGALS framework, which follows the DGALS MoC, should support the following features as guidelines:

1. There are both synchronous and asynchronous behaviors which are available in the conventional GALS MoC. Concurrent synchronous behaviors communicate with each other through signal broadcasting, so that all synchronous behaviors will have the same view of the signals. When asynchronous behaviors communicate with each other, there should be no shared data between them. Message passing should comply with pass-by-value semantics, which implies copying of messages. Synchronous behaviors within the same asynchronous behavior interact with each other by obeying the synchronous reactive MoC as in ESTEREL [Berry et al., 1983] and SystemJ that provide reactivity. The composition of asynchronous and synchronous behaviors is based on GALS MoC as used in CRP [Berry et al., 1993] and SystemJ.
2. A DGALS system can be distributed on networks of computational nodes. Asynchronous behaviors, which are not as tightly related as synchronous behaviors, can migrate within the DGALS system, according to the concept of weak mobility in DGALS MoC. Mobility also provides DGALS systems capability of fault tolerance and recovery, such as re-activating the backup asynchronous behavior at the same node or other nodes.

3. The DynamicGALS framework will provide programming interface, as part of the libDGALS, along with the run-time environment, to support communication, activations, and termination of asynchronous behaviors.
4. The libDGALS will be implemented in general programming languages, C in the current implementation, to support legacy code compatibility.
5. Last but not least, being based on formal MoC, the DynamicGALS framework opens the door to verifying DGALS systems with techniques used in the adopted MoCs including the synchronous reactive model, CSP, GALS model, and pi-calculus [Milner, 1999].

In the following, general features of the libDGALS are presented by a few small examples to illustrate the main properties of the DynamicGALS framework.

6.3.1 From libGALS to libDGALS

The static GALS systems created using libGALS can exploit only multicore processors and do not support distributed platforms. DynamicGALS framework, which is centered on libDGALS, evolves to allow exploiting both multicore and large distributed architectures. The static GALS systems in libGALS lack properties such as fault tolerance, mobility of code, and dynamic creation of behaviors, which essentially makes them very domain-specific. The libDGALS approach extends the static libGALS API (available in Chapter 4) with the goal that the DynamicGALS framework would make a good alternative to the general purpose concurrent libraries (such as pthreads).

libDGALS inherits basic design entities and objects introduced in the libGALS, including ‘clock domain’ (CD, as a group of synchronous behaviors, each CD is asynchronous to other CDs), ‘reactions’ (synchronous behaviors), ‘signals’ (means of communication between reactions in the same CD), and ‘channels’ (used for communications between reactions of different CDs). These elements are basic building blocks used to construct DGALS systems.

6.3.2 Structure of DGALS systems in the framework

The DynamicGALS framework allows the design of ‘DGALS systems’ that consist of multiple ‘DGALS programs’, which run on any core or computation node in a distributed (networked) system. A DGALS program can consist of one or more clock domains, which can be static (permanent for the system lifetime) or dynamic (non-permanent). Dynamic creation of CDs is supported through ‘CD plug-ins’, or ‘plug-ins’ for short. CD plug-ins encapsulate the body of the clock domains, reactions, channels, signals, and all other information necessary to create a CD and are instantiated upon activation. A plug-in is basically a library that can support ‘dynamic loading’, for example, a shared object (.so files) on Linux (or Unix-like) systems and a dynamic linking library (.dll files) on a Windows system. A plug-in must be defined and initialized before it can be used to create a new instance of the CD. Furthermore, a CD plug-in can be subsequently used to instantiate one or more CDs. Each CD created from the same plug-in can be customized according to ‘CD configurations’.

A designer defines the DGALS system, its DGALS programs, CDs and reactions, using the libDGALS API. Some of these API calls establish run-time data structures, while others are used to implement creation of new CDs, communication between CDs, as well as CD mobility.

6.3.3 Programming interface provided by libDGALS

Table 6.1 shows the descriptions of the programming interface that support dynamic features. Static systems can still be created with the programming interface inherited from libGALS. A DGALS program must be initialized by using the createDGALSProgram, and must be started by using startDGALSProgram. The CDPlugin macro is used to define the scope of a CD plug-in, and initPlugin is used for initializing the required data structure before the CD is instantiated. CD configurations, such as an identifier given to a newly activated CD, can be created with createCDConfiguration and extended via addCDConfiguration. Arguments passed to the activated CD, which are used to perform computations, can be similarly created and extended by using createCDArgument and addCDArgument, respectively. Other available functions, getCDArgumentNum, checkCDArgument, and getCDArgument, are used within reactions to obtain the arguments passed to the CD. Both configurations and arguments are used by activateCD to activate an instance of a CD plug-in on the destination machine. Any active CD can be terminated by using the terminateCD.

Table 6.1: API to program dynamic GALS systems

Function name	Description
createDGALSProgram	Instantiate data structures of the DGALS program
startDGALSProgram	Start the DGALS program and its Listener
CDPlugin (macro)	Start a CD plug-in definition
initPlugin	Initialize the data structure when creating an instance of a CD plug-in. This API is called at the beginning of the plug-in definition.
createCDConfiguration	Initialize a CD configuration of the new CD instance to customize parameters used in the CD. Returns: pointer to the CD configuration
addCDConfiguration	Add an entry to the CD configuration. Arguments: 1. existing CD configuration 2. configuration entry (key) to append 3. the value of the configuration to append
createCDArgument	Initialize a list of arguments passed to new CD instance Returns: pointer to the argument
addCDArgument	Add an argument to the list. Arguments: 1. argument list to append 2. name of the argument 3. type of the argument 4. the actual argument to pass
getCDArgumentNum	Check the number of arguments passed to the created CD instance
checkCDArgument	Check the availability of an argument. Argument: Name of the argument Returns: 1 - available, 0 - absent
getCDArgument	Obtain the argument by providing the name of the argument
activateCD	Activate a CD from a CD plug-in. Arguments: 1. destination DGALS program, where the CD will reside 2. name of the CD plug-in 3. configurations passed to the activated CD 4. arguments passed to the activated CD Returns: success / fail to activate the CD
terminateCD	Terminate a running CD Arguments: the name of the CD to terminate

A programmer must provide functions to serialize/de-serialize data used by a channel used for communication between CDs. These functions are called serialization and de-serialization functions. Table 6.2 lists the prototype names of these functions. Data to transfer are serialized/de-serialized with desired interpretations by design.

Table 6.2: Serialization and de-serialization functions

Function name	Description
serialize_data-type	Serialize function to encode data to a byte stream. Argument: the data to send Returns: unsigned char stream
deserialize_data-type	De-serialize function to convert a byte stream to the typed data. Argument: unsigned char stream Returns: reconstructed data

6.3.4 Simple examples to model dynamic behaviors

This section gives simple examples to familiarize the reader with the DynamicGALS framework and to present the system-level design features. Figure 6.1 and its corresponding DGALS code in Listing 6.1 show an example of CD instantiation and reactivity. The CD ‘cd1’ instantiates ‘cd2’ and ‘cd3’ on the ‘local DGALS program’ (named 192.168.1.1:1111) and a ‘remote DGALS program’ (named 192.168.1.2:1111), respectively, depending upon the value of the input signal cd1s1 received from the environment.

In Listing 6.1, firstly the required header-file (line 1) containing all the DGALS function definitions is included. The data structure ‘CDInfo’ is defined (lines 2-7) to hold information carried by the signal ‘cd1s1’. Input and output functions used by cd1 are defined (lines 8-9) and used to communicate with the environment to cd1. The ‘reaction function CD1R1Reaction’, which is the functional definition of reaction ‘cd1r1’, is declared on lines 10-25. cd1r1 firstly initializes by setting up argument (lines 11-13), and then waits for an incoming signal *cd1s1* (lines 14) and reads its value (line 15). Next, CD configurations are buit, which include the IP addresses, CD names, etc; indicating where the new CDs need to be instantiated (lines 16-17). Upon the *activateCD* (lines 18 and 21), the run-time environment activates a new CD instance on the correct physical machines with the CD configurations. The rest of Listing 6.1 (lines 25-31) shows how a CD is established when being activated from a CD plug-in, starting from the declaration of the scope of the plug-in (line 25), initialization of the plug-in (lines 26-27), instantiation of the clock domain, reaction and signal (lines 27-29), and finally the execution of the CD (line 30).

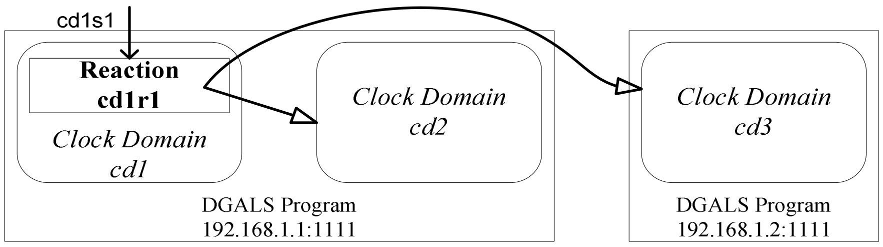


Figure 6.1: CD instantiation

Listing 6.1: CD instantiation and reactivity

```
1   #include "libDGALS.h"
2   typedef struct CDInfo {
3     char* progName;       // destination DGALS program
4     char* CDName;         // CD to activate
5     char* config;         // configurations of the activating CD
6     struct CDInfo *next;  // next entry
7   } CDInfo;
8   void IF(clockdomain CD) { ... }   // function to obtain input
9   void OF(clockdomain CD) { ... }   // function to generate output
10  REACTION_FUNCTION(CD1R1Reaction) {
11    initReaction();
12    signal cd1s1 = (signal)getArgument(1);
13    endInitReaction();
14    await(cd1s1);
15    CDInfo cds = value(cd1s1);      // read value of input signal
16    Configuration configCD2 = createCDConfiguration();
17    addCDConfiguration(configCD2, "CD.name;CD.rename", cdInfo->config);
18    activateCD(cds->programName, cds->clockDomainName, configCD2, 0);
19    cds = cds->nextCD;              // read next entry
20    .....
21    activateCD(cdInfo->programName,
22    cdInfo->clockDomainName, configCD3, 0);
23    endReaction();
24  }
25  CDPlugin {
26    initPlugin();
27    clockdomain cd1 = createClockDomain(IF, OF, "cd1", 0, 0);
28    signal cd1s1 = createSignal(cd1);
29    reaction cd1r1 = createReaction(cd1, CD1R1Reaction, 1, "cd1r1", 1, cd1r1);
30    startClockDomain(cd1);
31  }
```

Another simple example that demonstrates fault-tolerance capabilities is shown in Figure 6.2 and Listing 6.2. Only the important code segments are shown. In Figure 6.2, there are two CDs, ‘cd4’ and ‘cd5’, running on two different physical machines. Reaction ‘cd4r2’ keeps a check on the health of cd5, by receiving value (acting as heart beats) sending from cd5 through channel ‘ch2’ (line 21), and sending the result through signal ‘sSenderAlive’ to reaction ‘cd4r1’ (line 22), described in ‘CD4R2Reaction’. If cd5 dies, in the sense that sSenderAlive is not received in a certain time (maximum allowable number of ticks, lines 8-10), reaction cd4r1 activates a new instance of cd5 on the remote DGALS program (named 192.168.1.2:5555) and notifies cd4r2 to re-initialize (by sending sRestartRecv) the channel communication on ch2 (lines 11 and 12 respectively). The implementation of this behavior is shown in Listing 6.2.

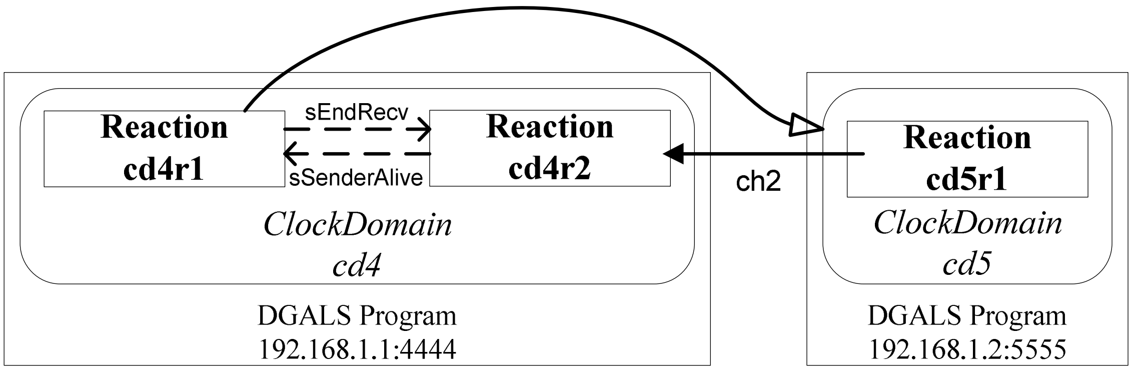


Figure 6.2: Fault tolerant systems designed in DGALS

The two simple examples presented above can be combined in a plethora of different ways to allow the designing of robust systems with the ease of describing reactivity and communication with the physical environment, the synchronous (reactions) and asynchronous (clock domains) concurrency, communication between the concurrent entities (reaction to reaction, channel to channel), weak code mobility and dynamic process forking and channel instantiation.

Listing 6.2: DGALS program implementing fault tolerance

```
1   REACTION_FUNCTION(CD4R1Reaction) {
2     initReaction();
3     signal sSenderAlive = (signal)getArgument(1);
```

```
    signal sEndRecv = (signal)getArgument(2);
    endInitReaction();
    int tickCount = 0;
    while(1) {
      if(present(sSenderAlive)) { tickCount = 0; }
      else tickCount++;
      if(tickCount == MAX_TICK_RESP) {
        activateCD("192.168.1.2:5555", "cd5", 0, 0);
        emit(sRestartRecv, 0);    // abort the current receive
      }
      pause();
    }
    endReaction();
  }
  REACTION_FUNCTION(CD4R2Reaction) {
    while(1) {
      strongAbort(sRestartRecv) {
        int value2; receive(ch2, value2, int);
        emit(sSenderAlive, 0);  // to inform cd4r1 aliveness
      }
      endAbort(sRestartRecv);
      pause();
    }
    endReaction();
  }
  CDPlugin {
    channel ch2 = createChannel("ch2", SenderCD, receiverCD);
    reaction cd4r1 = createReaction( /* arguments omitted */ );
    reaction cd4r2 = createReaction( /* arguments omitted */ );
    .....
```

6.3.5 DGALS programs and the run-time environment

Figure 6.3 illustrates a basic view of a DGALS system consisting of three DGALS programs running on Machine 1 (DGALS program 1 and 2) and Machine 2 (DGALS program 3), respectively.

As mentioned previously, the DynamicGALS framework provides libDGALS for programming, and a run-time environment for executions of the DGALS system that contains one or more DGALS programs. DGALS programs are responsible for: (1) managing the dynamic behavior of the CDs, (2) the mobility of CDs, (3) communication between CDs, and (4) implementing the overall DGALS MoC. Each DGALS program consists of the following:

1. Static linked libraries that support execution of the DGALS program, or dynamic linking libraries which are available and managed by the operating systems (OS) on the execution platform.

2. A local storage area that stores CD plug-ins. Local storage is generally governed by the file systems of the underlying OS.
3. Configurations of the DGALS program (or *program configurations*), which describes the location of the local storage, the list of CDs to be activated at startup, and the network port (a specific port number) that binds the essential communication to the underlying physical layer.
4. CD Configurations of the activated CDs. The DGALS program holds configurations of the running CD instances to manage creations and terminations of CDs.
5. Listener, which is a helping thread, and is invisible to the programmer. Listener is responsible for creating clock domains and channels according to the program configurations. Listener is also used to coordinate communication via channels between CDs (within the same DGALS program or between different DGALS programs). Mobility of CDs is also governed by Listener.

Channels are means of communication between reactions of different CDs. To establish such links, handshaking is first carried out by Listener, and is implemented with TCP/IP illustrated as point-lines in Figure 6.3.

Handshaking can occur on the same Listener, if both sending and receiving CDs are of the same DGALS program, such as the channel establishment between reactions of CD_{11} and CD_{12}. On the other hand, different Listeners will be involved if CDs are within different DGALS programs (CD_{21} to CD_{12}, and CD_{11} to CD_{31}), regardless of whether DGALS programs are running on the same machine or not.

Once the communication links are established, message passing takes place to perform the actual communications. There are two implementations of message passing: 'shared-memory' and 'TCP/IP' based. When shared-memory is used, messages are deep copied, through provision of serialization and de-serialization functions operating on the shared-memory. The shared-memory approach is adopted when both parties of a channel are in the same DGALS program, such as CD_{11} and CD_{12}. When TCP/IP is used, messages are serialized and sent to the receivers which reconstruct the original message through de-serialization functions. This approach is used when shared-memory is not available between different DGALS programs.

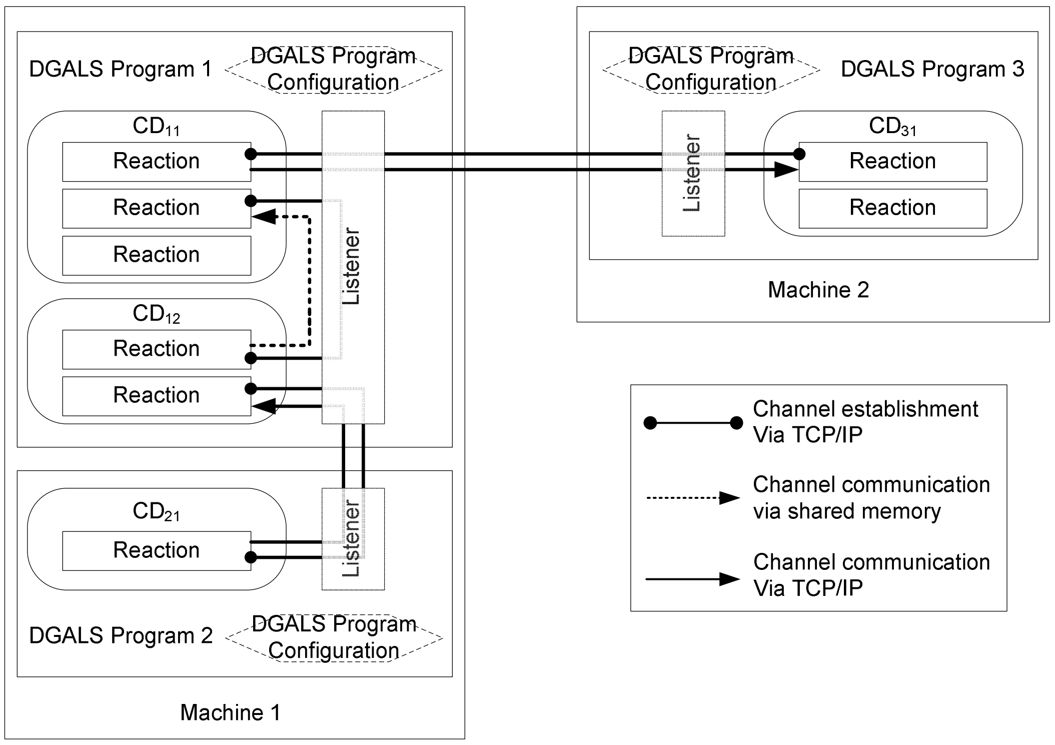


Figure 6.3: Channel implementation in a DGALS system

6.4 A complete DGALS system: dynamic Sieve

6.4.1 Dynamic sieve of Eratosthenes: prime number generation

Figure 6.4 illustrates a dynamic sieve of Eratosthenes (dynamic Sieve, or ‘Sieve’ for short), which illustrates the use of the DynamicGALS framework. Sieve is a DGALS system, which consists of three DGALS programs to calculate all the naturally occurring primes. Figure 6.4 shows only the calculation of primes up to six, because this suffices to explain the major design concepts and paradigms. A more complex example could have been chosen, but that would distract from presenting the features of the DynamicGALS framework.

Sieve consists of five CDs, ‘Generator’, ‘Shifter’, ‘Popper’, ‘Filter’, and ‘Printer’ running concurrently and asynchronously, each at their own speeds (logical ticks). In Figure 6.4, Shifter consists of three synchronous reactions, ‘PrimeShifter’, ‘ActivatePopper’, and ‘ActivateFilter’, respectively. These reactions communicate with each other using signals.

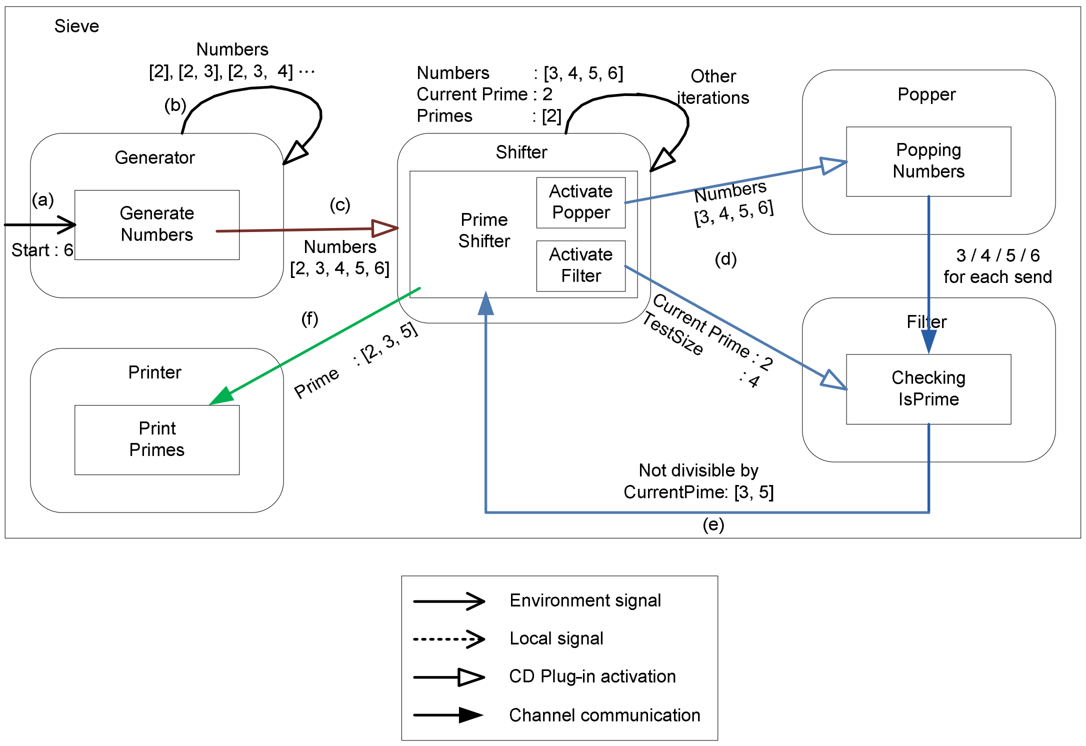


Figure 6.4: Dynamic sieve of Eratosthenes designed in the DynamicGALS framework

Sieve in Figure 6.4 is dynamic. At program startup, Generator and Printer are running, waiting for an incoming ‘start’ signal, as shown in Figure 6.4 (a), which determines the upper bound within which the primes need to be discovered. This bound is 6 in this example. Generator, upon reception of the start signal produces the set of natural numbers from 2 through to 6. This production is carried out using dynamic recursion of Generator. Each instantiation of Generator produces a natural number and adds it to the set ‘Numbers’, as (b). Next, Generator activates another instance of itself, passing the set Numbers as an argument. This dynamic recursion continues until the complete set of natural numbers smaller than the given upper bound is built. Generator then activates Shifter, which will now iteratively find the primes from within the generated natural-number set, as shown in Figure 6.4 (c). Shifter itself instantiates two new CDs, Popper and Filter, shown in Figure 6.4 (d). The first prime from the set (in this case, 2) is sent to Filter and the rest are passed onto Popper. Popper communicates with Filter by sending one number at a time through a dedicated channel. Filter extracts any number which is not divisible by the current filtering prime (2, as mentioned). The extracted set of numbers, containing 3 and 5, is passed to Shifter through another channel as shown in Figure 6.4 (e). Shifter puts 3 into the list of primes, by shifting out the first element of the set (contains 3 and 5) received from Filter previously. A new pair of Popper and Filter is instantiated again, by Shifter again, for the next iteration until the examining set in Shifter is 'null'. Finally, all the discovered primes are set to Printer, as illustrated in Figure 6.4 (f), which 'pretty prints' the discovered primes. Sieve highlights a number of features of the DynamicGALS framework:

1. *Reactivity* and *data fusion*: first of all, the DynamicGALS framework provides the explicit mechanism to capture signals coming in from the environment (e.g., signal named 'start' in Figure 6.4). This attribute directly satisfies support for data fusion capabilities. This concept of reactivity is inspired by ESTEREL [Berry, 1993].
2. *Hierarchical concurrency* and *safe message passing*: The CDs, as synchronous islands, allow an easy way to express tightly coupled concurrent behaviors (called synchronous parallel reactions or just reactions in this case). Reactions in different CDs communicate with each other over point-to-point channels using CSP-style rendezvous [Hoare, 1978], thus the blocking send and receive, which in turn guarantees data delivery.
3. *Dynamic behaviors* and *Robustness*: The DynamicGALS framework allows the instantiation of new CDs at run-time (dynamic creation); it also allows the destruction (termination) of CDs at run-time. The channels associated with dynamic CDs are also created at run-time. The formal DGALS MoC along with dynamic creation and destruction provide fault tolerance capabilities. For example, an error in a certain part of a large design can be

corrected and that portion restarted without affecting the rest of the running system. New physical sensors and other units can be loaded at run-time.

4. *Abstraction of execution platforms* and *topologies*: It should be noted that the programs developed using the DynamicGALS framework are detached from the underlying physical execution layer. For example, Sieve in Figure 6.4 is designed without any concern for the underlying execution and communication architecture. In fact, the same Sieve example can be implemented on hosts of different heterogeneous execution and communication platforms. This separation between design and physical implementation provides an abstraction layer, which essentially speeds up the development, because the underlying physical layer and the software model can be developed in parallel. More importantly, a DGALS program is immediately ready for execution on a single processor system, but the same specification can run on different execution platforms without any change. Also, the aforementioned separation increases fault tolerance and recovery capabilities, as the designed model can be changed at run-time without affecting the underlying physical implementation layer and vice-versa.

Other features, which further enhance the design capabilities of the DynamicGALS programming framework such as weak code mobility, are not presented in Figure 6.4. Such capability, closely related to the underlying physical architecture, is explained in the next section.

6.4.2 Distributed dynamic Sieve

In this section the implementation of the dynamic Sieve model on a heterogeneous and distributed physical execution and communication layer is presented. The purpose of this description is to demonstrate the features of libDGALS on distributed architectures.

Figure 6.5 is an abstract representation of the dynamic Sieve. There are three physical machines as computation nodes, connected via network (LAN/WAN). A single DGALS program runs on each of these three different machines.

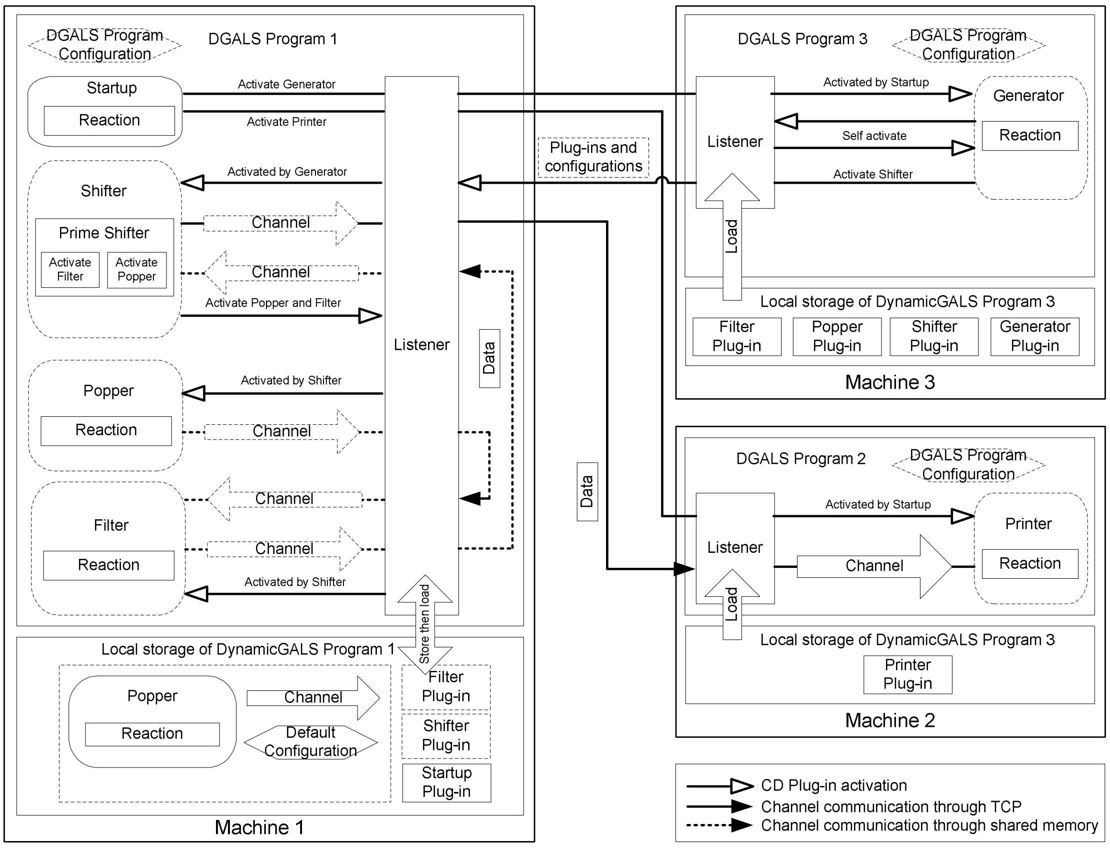


Figure 6.5: The distributed dynamic Sieve

As shown in Figure 6.5, clock domain ‘Startup’ activates Generator of the Sieve example on ‘DGALS program 3’ executing on ‘machine 3’. Similarly, Printer is initially activated on ‘DGALS program 2’ of ‘machine 2’ by Startup. Once the generation of the natural number set is complete through recursive self-activation of Generators, Generator activates Shifter on DGALS program 1 running on machine 1. Shifter then instantiates Popper and Filter CDs within the same DGALS program.

Figure 6.5 shows the transfer of the CD plug-ins (Shifter, Popper, and Filter in this case) along with their configurations, from DGALS program 3, to Listener of the DGALS program 1, shown in the dotted box. Note that Generator and Printer plug-ins need not to be sent from the DGALS program 1 as they are available on the destination DGALS programs. Filter and Popper communicate with each other using channels on the same computation node (machine 1) via Listener, to extract the primes. Once the final set of primes is obtained, Shifter passes this set onto Printer to pretty print the set.

The data sent through channel require serialization at the sending side (Shifter) and de-serialization at the receiving side (Printer). In

Figure 6.5, the CDs in solid round rectangles represent the CDs instantiated at program startup, i.e., they represent static CD invocations, while the dotted ones show the CDs that are invoked at run-time. Similarly, channels created at startup and run-time follow the same representation. Thus, Generator and Printer are instantiated at program startup, while Shifter, Popper, and Filter are instantiated dynamically at run-time. Finally, it should be noted that while the Printer is alive throughout the application lifetime, the remaining CDs do not and they are terminated when they are not needed.

6.4.3 Implementation of the dynamic Sieve

A DGALS program can be described as shown in Listing 6.3. In practice, a DGALS program consists of initialization of other codes, which will be used by the DGALS program, e.g. device drivers. The *createDGALSProgram* and *startDGALSProgram* (lines 4-5) are called to initialize the essential data structures for the program and Listener, followed by the start of the program.

Listing 6.3: A simple DGALS program

```
1   #include "libDGALS.h"   // required to use the libGALS API
2   void main() {
3     ......                // initialization for non-DGALS program, e.g. driver
4     createDGALSProgram(); // setup data structures and Listener
5     startDGALSProgram();  // start the DGALS program
6   }
```

In the DynamicGALS framework, a CD can be created dynamically only if it is instantiated from a CD plug-in. The construction of a CD plug-in follows a bottom-up strategy and consists of the following:

1. Reaction functions, from which reactions will be instantiated, describe the functionalities of the reactions. One reaction function can be used to create more than one reaction of the same clock domain.
2. Definition of the CD plug-in, which is composed of reactions, signals, and channels. When a plug-in is activated, the corresponding elements are instantiated in the DGALS program.

3. Default CD configuration of the plug-in. Parameters of the configuration include the names of the CD and used channels. These parameters are hard-coded as the naming reference which will be overridden during plug-in activation. Reaction and signal names are encapsulated by the CD as they cannot be accessed by other CDs, and are thus not included in the default configuration.

Listing 6.4 shows CD Startup, which initializes the Sieve example. In the reaction function ‘StartupReaction’, Startup (lines 6-10) activates Generator and Printer. Relationship of the ‘requester CD’ (the CD that activates the other CD, for example the CD Startup) and the ‘responder’ CD (the CD to be activated, such as Generator and Printer) are established when invoking the ‘activateCD’ **(**lines 8 and 9). The essential information to activate a CD is given as follows:

1. The name of the destination DGALS program. A DGALS program name is a combination of the machine (where the DGALS program executes) name and the port bind to the Listener of the DGALS program. For example, DGALS program 3 running on machine 3 is named as ‘machine3:12222’.
2. The name of the CD plug-in to activate. This is also the file name of the CD plug-in. For example, plug-in Generator will be stored in Generator.so on Unix-based systems.
3. The CD configuration used for the activation. CDs and channels are means of describing asynchronous behaviors and communications in the DGALS system. Names of CDs and channels are unique to differentiate them from others. The configuration consists of name mappings of both CDs and channels, from the hard-coded reference, to the assigned unique name.
4. The argument passed to the CD. More than one CD can be instantiated from the same CD plug-in. Each instance of the CD might require different information, depending on the nature of the computation performed by the CD. Such information is passed as arguments to the activated CD. The difference between configuration and argument to a CD is that configuration

over-rides the existing hard-coded information, while arguments are created on an as-required basis.

Thus, during execution of the activateCD calls, Generator and Printer are activated in DGALS program 3 and DGALS program 2, respectively. A plug-in is defined within the scope of the macro ‘CDPlugin’ (lines 13-19). ‘initPlugin’ is called to set up data structures of the plug-in (line 14) followed by the creation of the CD (line 16), reaction (line 17), and starting of the CD (line 18).

Listing 6.4: The StartupCD of dynamic Sieve

```
1   #include "libDGALS.h" // required to use the libDGALS API
2   // input and output functions, to communicate with the environment
3   void IFC0(void) {......}
4   void OFC0(void) {......}
5   // the code for the startup reaction.
6   REACTION_FUNCTION(StartupReaction) {
7     ......
8     activateCD("machine3:12222", "Generator", 0, 0);
9     activateCD("machine2:12222", "Printer", 0, 0);
10    endReaction();
11  }
12  // definition of the Startup CD plug-in
13  CDPlugin {
14    initPlugin(); //setup data structures and Listener
15    // elements of the plug-in
16    clockdomain Startup = createClockDomain(IFC0, OFC0, "cdStartup", 0, 0);
17    createReaction(Startup, StartupReaction, 1, "rStartup", 0);
18    startClockDomain(StartupCD);
19  }
```

Listing 6.5 describes the Generator CD plug-in, which is activated in Listing 6.4. It follows the same design approach: to include the required header files (lines 1-3), in which libDGALS API, user defined data structure, and constants are available. This is followed by the definition of the reaction function ‘GeneratorReaction’ (lines 4-56). Arguments passed to a reaction can be obtained by calling ‘getArgument’ (line 6). A reaction function has a set of local variables (lines 8-10) for carrying out internal algorithms, or to hold values from signals. The value of a signal can be obtained with the use of ‘value’ (line 15).

The number of arguments passed to a CD can be accessed with ‘getCDArgumentNum’ (line 11). A return value of zero indicates that no argument was passed to the plug-in. Arguments passed to an activated CD can be obtained via ‘getCDArgument’. This takes the names of the arguments (e.g. Numbers or start) and the corresponding types (IntegerSet or int) as shown in lines 22-23. Arguments sent to CDs are constructed using ‘createCDArgument’ and ‘addCDArgument’ (lines 33-35 and 47-49).

Configurations provided to a CD, which are prepared through using ‘createCDConfiguration’ and ‘addCDConfiguration’ (lines 37 and 41 respectively), are in the form of strings (lines 38-40). Both configurations and arguments are used when issuing ‘activateCD’ (line 44 and 52).

Listing 6.5: The Generator of the dynamic Sieve

```
1    #include "libDGALS.h"
2    #include "IntegerList.h" // user defined typed used in Sieve
3    #include "Sieve.h"       // define constants such as DGALS program name SHIFTER_DP
4    REACTION_FUNCTION(GeneratorReaction){
5      initReaction();                              // initializing this reaction
6      signal start = (signal)getArgument(1)        // get argument-to-reaction
7      ......
8      int start = 1;                               // default lower bound
9      int MAX = 17;                                // default upper bound
10     IntegerSet* Numbers = 0;                     // the natural number set
11     if(getCDArgumentNum() == 0) {
12       // no argument is given to this plug-in instance,
13       // therefore it is the first Generator
14       await(start);                              // wait for start signal
15       start = value(start);
16       start = start + 1;                         // allocate first number to the set
17       Numbers = (IntegerSet*)calloc(1, sizeof(IntegerSet));
18       Numbers->value = start;
19     }
20     else {
21       // get arguments passed to this plug-in instance
22       Numbers = getCDArgument("Numbers", IntegerSet);
23       start = getCDArgument("start", int);
24       start = start + 1;  // extend the set with new numbers
25       // working on the received arguments
26       ......
27     }
28     // pass the set of complete natural numbers to the Shifter
29     if(start == MAX)
30     {
31       int id = 1;
32    // create arguments passed to activate Shifter
33    Argument* argsToShifter = createCDArguments();
34    addCDArgument(argsToShifter, "Numbers", IntegerSet, Numbers);
35    addCDArgument(argsToShifter, "id", int, id);
36    // create binding configuration for Shifter
37    Configuration* cfgSft = createCDConfiguration();
38    char* Sftrs = (char*)calloc(1, sizeof(char)*strlen("Shifter;Shifter")+4);
39    sprintf(Sftrs, "Shifter;Shifter%02d", id);
40    addCDConfiguration(cfgSft,"clockdomain.name;clockdomain.rename",Sftrs);
41    // other configurations
42    ......
43    // activate Shifter on DGALS program whose name is define in SHIFTER_DP
44    activateCD(SHIFTER_DP, "Shifter", cfgSft, argsToShifter);
45   }
46   else {
47    Argument* argsToGenerator = createCDArgument();
48    addCDArgument(argsToGenerator, "start", int, start);
49    addCDArgument(argsToGenerator, "Numbers", IntegerSet, Numbers);
50    // configurations to name new instance of Generator
51    ......
52    activateCD(GENERATOR_DP, "Generator", cfgGen, argsToGenerator);
53   }
54   pause();                // finish a logical tick
55   endReaction();          // end of the reaction
56  }
57  CDPlugin {
58   // similar to Listing 6.1 to create CD, reactions, signals, and channels
59   ......
60  }
```

Listing 6.6 shows the partial implementation of Shifter and focuses on support for reactivity and synchronous parallel reactions within a CD. These features can be implemented using libGALS API, illustrating that libDGALS is compatible with libGALS. A reaction is initialized via the ‘initReaction’ (line 4). A reaction can obtain the arguments passed to it (line 5-9) by calling ‘getArgument’ It is followed by the end of the initialization block of the reaction function, by calling ‘endInitReaction’ (line 11). ‘checkCDArgument’ is used to check the availability of an argument passed to this CD on line 14. The corresponding argument can be extracted with the ‘getCDArgument’ function (line 15). Thus the value of ‘id’ is obtained from the plug-in argument and is assigned to a valued signal (signal_id) by calling ‘emit’ (line 18), which in turn makes it visible to all the synchronous parallel reactions (‘ActivateFilter’ and ‘ActivatePopper’, which are instantiated from reaction functions ‘ActivateFilterReaction’ and ‘ActivatePopperReaction’) running within Shifter. Child reactions can be forked (line

19) from the parent reaction, which will be blocked until all of its child reactions jointly finish execution (line 20). A reaction can communicate with other reactions in a different CD using ‘send’/’receive’ that operate on channels (lines 25 and 37). Child reactions (e.g. ActivateFilter and ActivatePopper) are able to receive signals emitted from the parent reaction (e.g. PrimeShifter) because both children and parent reactions are in the same CD. The value of a signal can be obtained by calling ‘value’ (line 46). Information such as the identification (names) of a channel, its sending CD, and its receiving CD are predetermined (as part of the default configuration of the plug-in, as lines 58-59) and can be re-assigned through configurations when activating the plug-in.

Listing 6.6: Shifter of the dynamic Sieve

```
1      // SHIFTER_DP, POPPER_DP, FITER_DP, and PRINTER_DP are string constants
2      // representing the names (addresses with bind ports) of DGALS programs
3      REACTION_FUNCTION(PrimeShifterReaction) {
4        initReaction();
5        channel cFilterToShifter = (channel)getArgument(1);
6        channel cShifterToPrinter = (channel)getArgument(2);
7        reaction rActivatePopper = (reaction)getArgument(3);
8        reaction rActivateFilter = (reaction)getArgument(4);
9        signal signal_id = (signal)getArgument(5);
10       ......
11       endInitReaction();
12       ......
13       int id = 0;
14       if(checkCDArgument("id") == 1)
15         id = getCDArgument("id", int);
16       // processing numbers to be used by Popper and Filter
17       ......
18       emit(signal_id,id);
19       ......
20       // fork 2 child reactions wait them for completion
21       fork(rActivatePopper); fork(rActivateFilter);
22       join(rActivatePopper); join(rActivateFilter);
23       // receive from Filter via a channel
24       IntegerSet* listOfNonDivisibles = 0;
25       receive(cFilterToShifter, listOfNonDivisibles, IntegerSet);
26       // add the values from the received to gens
27       if(listOfNonDivisibles != 0) {
28         // re-iterate the process by activating another instance Shifter until
29         // the complete set of primes is found (no non-divisibles left to process)
30         Argument* argsToShifter = createCDArgument();
31         addCDArgument(argsToShifter, "Numbers", IntegerSet, listOfNonDivisibles);
32         ......
33         activateCD(SHIFTER_DP, "Shifter", configShifter, argsToShifter);
34       }
35       else {
    // send the final set of primes to Printer
    send(cShifterToPrinter, Prime, IntegerSet);
  }
  endReaction();
}
REACTION_FUNCTION(ActivateFilterReaction) {
  initReaction();
  signal signal_id = (signal)getArgument(1);
  ......
  endInitReaction();
  int id = value(signal_id);
  ......
  // activating Filter with configurations and arguments
  activateCD(FILTER_DP, "Filter", configFilter, argsToFilter);
  endReaction();
}
// reaction function to activate Popper
REACTION_FUNCTION(ActivatePopperReaction) { ... }
CDPlugin {  // definition of the CD plug-in
  // similar to Generator
  ......
  // create channel to transfer non-divisibles of current iteration
  channel cFilterToShifter = createChannel(
      SHIFTER_DP"Filter", PRINTER_DP"Shifter", "cFilterToShifter");
  ...... // other channels or so
}
```

6.4.4 Configurations of a DGALS program

When a CD is activated, it has to be accompanied by the ‘CD configuration’, which can be either ‘remote configurations’ or ‘local configurations’. The remote configurations are these used to activate CDs which are specified at run-time with ‘activateCD’ calls. Local configurations, on the other hand, are created statically to activate CDs at the DGALS programs start up. As mentioned in Section 6.3.5, each DGALS program is equipped with a dedicated set of ‘program configurations’ which specify parameters such as the port (number) used by Listener. Local configurations of CDs are considered as part of the DGALS program configurations. Configurations of each DGALS program are loaded when the DGALS program starts, and are stored in the XML format.

Listing 6.7 shows an XML configuration of DGALS program 1 on machine 1 shown in Figure 6.5. Each key-value pair represents settings for the specified compartment, or a scope of a component. The <port> (line 1) indicates the port number on which the Listener of this DGALS program will listen. The <timeout> node (line 2) is used as the time-out value for Listener when participating in channel communication, and plug-in activations as explained in detail in Section 6.5. The configuration of a statically loaded CD starts with <plugin> (line 3), along with the name of the plug-in (Startup, which is the name of the plug-in) to load. Each plug-in consists of one CD and one or more channels. The CDs and channels are visible system-wide, and each one needs to have a unique name. A name re-mapping of a CD from the referenced name (given as cdStartup in the 'createClockDomain' of Listing 6.4) to a globally system-wide unique name is provided, starting with <clockdomain> (line 4). The original CD name (cdStartup) within a plug-in is identified through the <name> and </name> pair (line 5). The re-mapped name is then provided (Startup) and wrapped between <rename> and </rename> (line 6). The name re-mapping of a CD, which is ended with </clockdomain>, is followed by </plugin> as the end of the CD configuration, on lines 7 and 8 respectively. A program configuration can have more than one plug-in section.

Listing 6.7: The XML configuration of the DGALS program 1

```
1    <port>12222</port>
2    <timeout>3</timeout>
3    <plugin>Startup
4      <clockdomain>
5        <name>cdStartup</name>
6        <rename>Startup</rename>
7      </clockdomain>
8    </plugin>
```

To show that CDs can be loaded statically at the beginning of the DGALS program in the Dynamic Sieve, CD Startup is removed from dynamic Sieve, shown in Figure 6.5, and the resulting dynamic Sieve in shown in Figure 6.6. In this case, both Generator and Printer will be required to be activated through the use of DGALS program configurations. In Figure 6.6, Printer is loaded on DGALS program 2 of machine 2, and a channel (named cShifterToPrinter, hard-coded as default configuration in Printer) is used to receive the resulting primes from Shifter. Thus, the DGALS program configurations, shown in Listing 6.8, detail the activation of Printer and the required name mappings of the channel used.

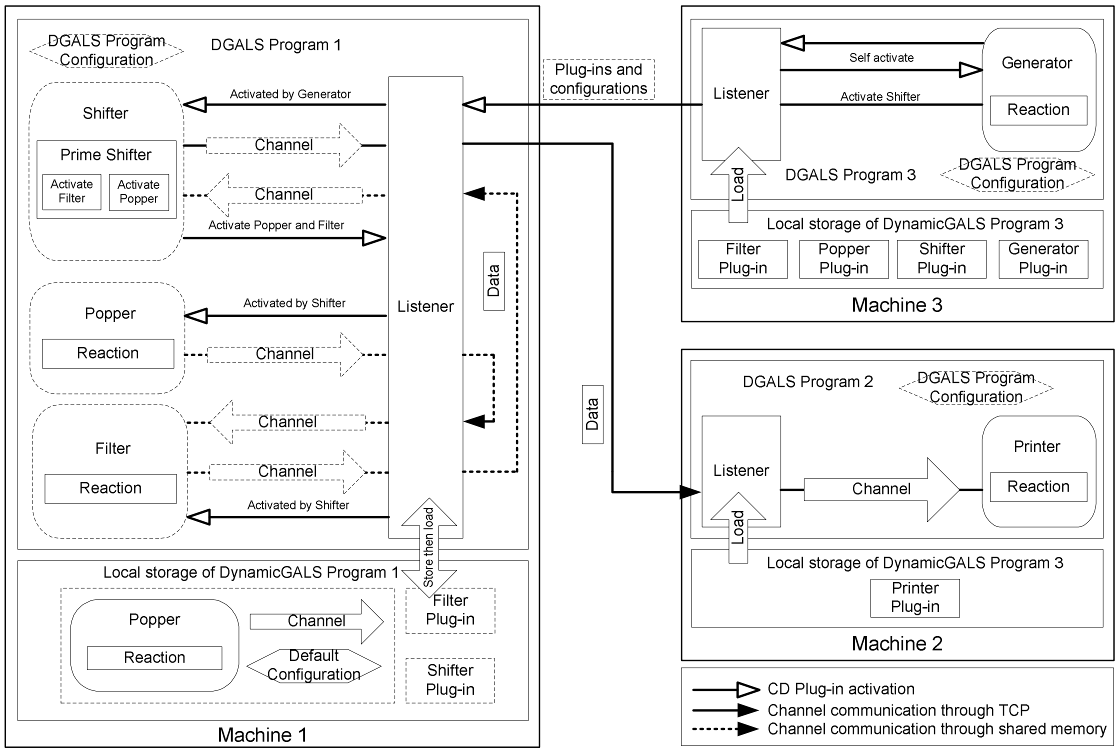


Figure 6.6: The distributed dynamic Sieve without CD Startup

Listing 6.8 follows the conventions of Figure 6.6. Similarly, name re-mappings of the channels are given between the <channel> and </channel> tags. The name of the channel (from the ‘createChannel’) in the plug-in is given and is followed by the re-mapped name of the channel. Since the sending and receiving parties of a channel are CDs, it is required to provide the correct CD names to link with the channel. <sender></sender> and <receiver></receiver> pairs are dedicated for this requirement. A CD name is in the format of ‘Machine:Port:CDName’, or just ‘CDNmae’ if running locally. For instance, ‘Machine1:12222:Shifter’ indicates that Shifter will be running within the DGALS program which binds port 12222 on Machine 1, whereas Printer01 will be executed locally. The order of the re-mappings for the CD and channels is not important as long as they are all listed.

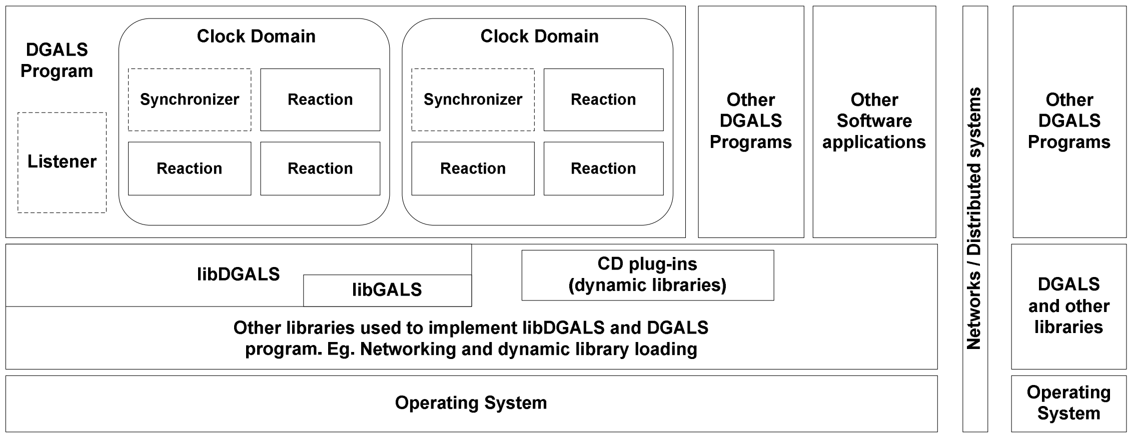
Listing 6.8: The XML configuration of the DGALS program 2

```
<port>12222</port>
<timeout>3</timeout>
<plugin>Printer
```

```
    <clockdomain>
      <name>Printer</name>
      <rename>Printer01</rename>
    </clockdomain>
    <channel>
      <name>cShifterToPrinter</name>
      <rename>ShifterToPrinter01</rename>
      <sender>Machine1:12222:Shifter</sender>
      <receiver>Printer01</receiver>
    </channel>
  </plugin>
```

6.5 The DynamicGALS framework implementation

The libDGALS in the DynamicGALS framework extends the libGALS detailed in Chapter 4. Figure 6.7 presents a high-level view of the library and run-time system provided by the DynamicGALS framework. DGALS programs are positioned within its run-time environment and communicate with other DGALS programs, locally or over the network. Listener is invisible to the programmer and supports dynamic creation/destruction of CDs, channel-based communication, and CD mobility within a DGALS program. Synchronizer is responsible for lockstep execution of reactions within a CD. Currently, all the concurrent entities, which include Listeners, Synchronizers, and reactions in Figure 6.7, are mapped to POSIX threads.


DGALS Program
Listener
Clock Domain
Synchronizer
Reaction
Reaction
Reaction
Clock Domain
Synchronizer
Reaction
Reaction
Reaction
Other DGALS Programs
Other Software applications
Networks / Distributed systems
Other DGALS Programs
libDGALS
libGALS
CD plug-ins (dynamic libraries)
Other libraries used to implement libDGALS and DGALS program. Eg. Networking and dynamic library loading
DGALS and other libraries
Operating System
Operating System
Figure 6.7: The programmers' perspective of the DynamicGALS framework

6.5.1 Data structures used by DGALS programs

Each DGALS program and each activated CD operate over a special data structure called ‘Run-time information’, which is used to book-keep the status of the program and consists of two parts, ‘ProgramData’ and ‘PluginInstance’. ProgramData contains the global view of the DGALS program, such as the unique program name and the names assigned to the activated CDs.

PluginInstance keeps a unique record of each activated CD instantiated from a CD plug-in. Each PluginInstance is assigned to a CD instance, and thus the CD plug-in allows multiple CDs instantiated from the same CD plug-in, which in turn enables code re-use at a coarser level of granularity. The data structures are complex and a complete explanation of each part is beyond the scope of this thesis. To achieve efficient implementation, instead of using interprocess communication (IPC) to operate on Run-time information, reactions, Synchronizers, and Listener are implemented as threads to share the Run-time information.

6.5.2 Reactions and Synchronizers

Reactions and Synchronizers are implemented in libGALS and their use is extended to libDGALS. A reaction is implemented as a thread whose execution body is defined by a reaction function. Multiple reactions can be spawned from the same reaction function to achieve code re-use at a finer granularity. Synchronizer is a special thread that manages reactions within a CD. Synchronizers are programmer invisible and are created at run-time by libDGALS when corresponding CDs are activated.

6.5.3 Listener

Listener is a special and dedicated thread created for each DGALS program, in charge of channel communication, CD activation, and CD termination. Listeners also communicate with those of other DGALS programs to achieve these functionalities. Communication between Listeners is accomplished in two phases: first, handshaking to establish the link, then the transference of actual information. Both are carried out by sending and receiving messages from one Listener to the other. Handshaking is via TCP/IP and transferring of information can be based on shared memory or TCP/IP depending on the topology of DGALS programs as described in Section 6.3.5. Messages contain ‘headers’ that include the source and destination of the DGALS programs, added by Listeners; Listeners can thus identify the underlying DGALS program as the sender or receiver of the messages by checking the headers. If a message is sent and received via the same Listener, this effectively implements a loopback so that shared memory is used to shorten the time of delivering messages. In this way, the programmer does not need to worry how the messages are sent and received. This, in turn, satisfies the requirement of ‘location transparency’. Current implementation divides messages operated upon by Listener into two groups: (1) those that represent channel communication and (2) those that represent plug-in activation/CD termination, respectively. Types of messages are also embedded in the headers of the messages. Listener spawns its own child threads to decode messages for each incoming connection to the DGALS program.

6.5.4 Scheduling of reactions, Synchronizers, and Listener

The scheduling of reactions is handled by the host operating system (OS) scheduler, which works closely with the Synchronizer. If a reaction is blocked due to a libDGALS API call, control is transferred to another reaction that is ready for execution. The interleaving of reaction execution and transfer of the processor control from one reaction to another is governed by the OS scheduler. However, a reaction cannot be scheduled to be executed unless it has the permission of its CD Synchronizer, which enforces lock-step execution of reactions, and hence the synchronous MoC within CDs. Scheduling strategies of different OSs only affect the execution sequence of reactions that do not have mutual signal dependencies. A reaction can run in parallel with reactions in the same CD, given that these are not blocked due to signal dependencies and if the execution platform allows it (e.g. on a multicore platform). Listener along its child threads, are scheduled by the OS in the same fashion as the asynchronous execution and activation of the CDs.

DGALS programs are multithreaded, the threads being implemented by using POSIX threads, i.e. pthread library [Nichols et al., 1996]. In DGALS programs, several kinds of threads are created as shown in Figure 6.7: (1) the main program thread, which in turn becomes the DGALS program, (2) reaction threads, (3) Synchronizer threads, (4) Listener thread, and (5) message-decoding threads for channel communications and CD activations/terminations.

Each reaction maps onto a single reaction thread. For example, if there are three reactions in a clock domain, three reaction threads will be created. One Synchronizer thread will be created for each CD. The Listener thread, one for each DGALS program, waits for incoming messages and spawns child threads to decode the messages sent to the Listeners. Reaction threads are terminated when a CD is terminated. The child threads of Listener threads terminate when messages are decoded and actions are performed. Listener thread terminates only when the DGALS program terminates.

Choosing between user-level or kernel-level threading libraries is application dependent. Since kernel-level threading maps each thread to processes of the OS, executions of these threads can benefit from the multicore architecture. This is suitable for parallelizing data computation in the reactions of either the same or different CDs. However, DGALS programs using kernel-level threading might suffer from performance drawback because of context switching at the kernel level, for systems with a minimal number of data computations. Using a user-level threading library can be seen as the remedy. However, such a DGALS program will not benefit from the multicore platform.

6.5.5 CD activation and termination

CD activations are governed by Listeners who start a handshaking prior to the instantiation of a CD plug-in. The CD activation is carried out between two CDs, which are called the ‘requester’ and the ‘responder’. The requester CD requires from another CD, the responder, to be activated by Listeners via ‘activeCD’. When one requester and the responder belong to the same DGALS program, it is equivalent to spawning a new plug-in instance locally. Listener handles the incoming messages used for the CD activation and changes the state of ‘PluginInstance’. A state variable in PluginInstance indicates the current state of the CD activation. Figure 6.8 shows the finite state machine (FSM) of the requester. To simplify the data structure, state naming is shared between the requester and the responder. This is illustrated in Figure 6.9, the FSM of the responder. However, requester and responder work on different copies of PluginInstance. The PluginInstance at the responder side is registered and permanent, while that at the requester side will be eliminated once the plug-in is activated.

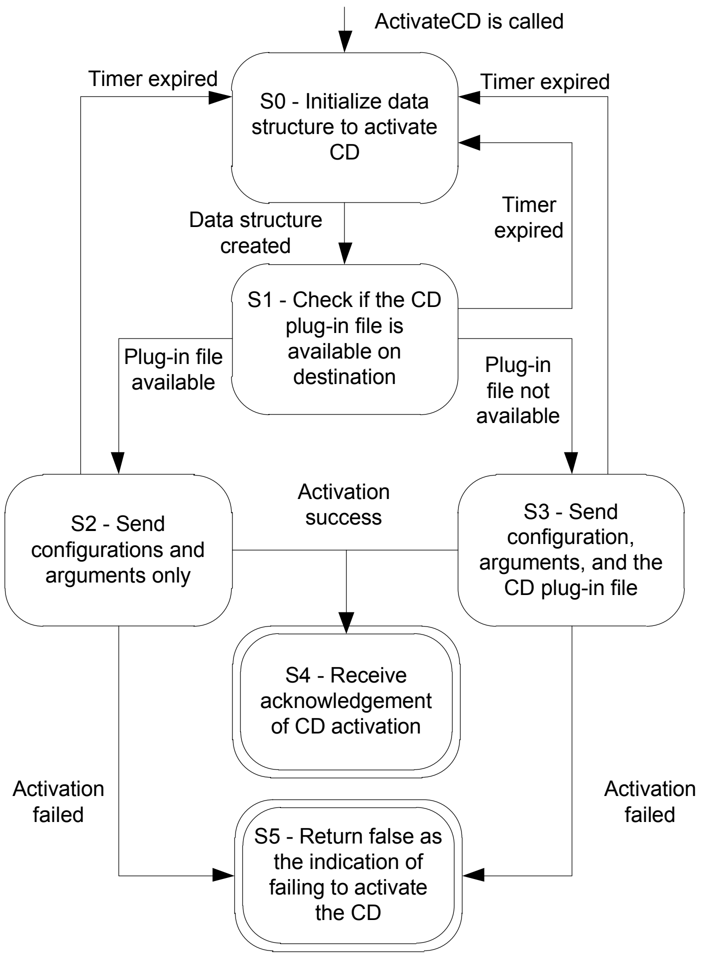

ActivateCD is called
Timer expired
S0 - Initialize data structure to activate CD
Timer expired
Data structure created
Timer expired
S1 - Check if the CD plug-in file is available on destination
Plug-in file available
Plug-in file not available
S2 - Send configurations and arguments only
Activation success
S3 - Send configuration, arguments, and the CD plug-in file
S4 - Receive acknowledgement of CD activation
Activation failed
Activation failed
S5 - Return false as the indication of failing to activate the CD

Figure 6.8: FSM of the requester

The activation of a CD requires configurations of the activated CD. This configuration contains information such as the activated CD's mapped name which is checked for any duplication by Listener. The requester will be notified if there is a naming conflict or the responder fails to be activated; for example if the existing name of a channel has been given to a new instance. CD termination can be requested by any CD by calling the *terminateCD*. CD termination follows the same state-based approach as CD activation.

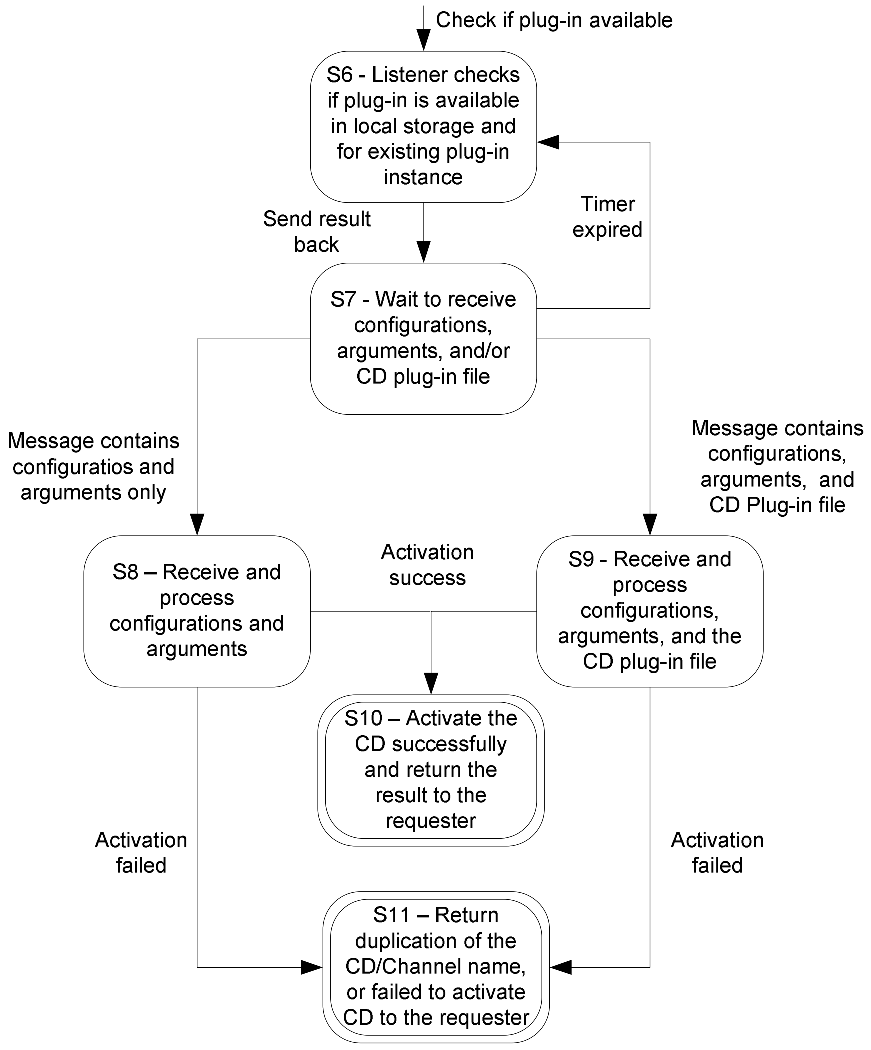

Check if plug-in available
S6 - Listener checks if plug-in is available in local storage and for existing plug-in instance
Send result back
Timer expired
S7 - Wait to receive configurations, arguments, and/or CD plug-in file
Message contains configuratios and arguments only
Message contains configurations, arguments, and CD Plug-in file
S8 – Receive and process configurations and arguments
Activation success
S9 - Receive and process configurations, arguments, and the CD plug-in file
S10 – Activate the CD successfully and return the result to the requester
Activation failed
Activation failed
S11 – Return duplication of the CD/Channel name, or failed to activate CD to the requester

Figure 6.9: FSM of the responder

6.5.6 Channel communication and rendezvous in libDGALS

The libDGALS in the DynamicGALS framework inherits the point-to-point, rendezvous-based communication mechanism from libGALS, which is semantically identical to one used in SystemJ [Malik et. al, 2010]. Communication in channels is similar in functionality to the CD activation mechanism described in Section 6.5.5, that is, based on handshaking and message passing. Data sent over channels are stored in different copies by means of deep copying. As mentioned in Section 6.3.5, two implementations are available: TCP/IP and shared-memory based.

In the TCP/IP implementation, data are serialized through the use of a serialization function (used by the ‘send’ API call), provided by the designer, for each data type. The serialized data are sent through TCP/IP as payloads along with headers inserted by Listeners, and are received by Listener at the DGALS program where the receiver CD resides. The ‘receive’ API call utilizes the de-serialization function and restores the data. On the other hand, in the shared-memory implementation, send API call serialized and de-serialize functions create a deep copy of data-to-send in the heap. A pointer to the copied data is used directly by the receive API call. No data transfer over the network stack is required in this case to reduce the workload of Listener effectively.

6.5.7 DGALS system over distributed systems

Different virtual topologies of CDs (not necessarily representing the underlying physical architecture) can be established by the designer allowing them to logically arrange the DGALS programs into DGALS systems based on convenience and practical requirements. Benchmarks in Section 6.6 present examples of partitioning strategies for DGALS systems into a number of DGALS programs running on different physical machines, effectively building virtual topologies. In general, design-space exploration is required to construct the most efficient topologies and partitioning of CDs.

6.6 Experimental results

A number of experiments with different examples and physical execution-platform setups to gauge the effectiveness of the DynamicGALS framework approach have been carried out. The benchmark set is shown in Table 6.3.

Table 6.3 shows the name of the application, followed by the name of the CD plug-ins used in the application. This is followed by the number of instances of those plug-ins created. The numbers of channels and reactions in each plug-in are also provided. The code size is given for each plug-in and the complete DGALS system. The Send-Receive example acts as a micro-benchmark, which gauges the efficiency of the fundamental CD instantiation and channel communication mechanism. The Sieve, which has been used as a running example throughout this chapter, has been coded in two different versions. The dynamic version is as shown in Figure 6.6, while the static version is created from the dynamic version after finding the overall number of CDs and channels instantiated in the lifetime of the dynamic version and instantiating all as static.

Table 6.3: Benchmarks selected for experimentation

Applications	CD plug-ins	Number of Instances	Number of channels	Number of reactions	Code Size (KB) Plug-ins total Size
Send Receive 141K					
	SendCD	1	1	1	8.8K
	ReceiveCD	1	1	1	9.2K
Sieve (prime < 17) static version 150K (Main program and plug-ins altogether)					
	Generator	16	2	1	
	Shifter	7	6	3	
	Popper	7	2	1	
	Filter	7	3	1	
	Printer	1	1	1	
Sieve (prime < 17) dynamic version 188.3K					
	Generator	16	0	1	13K
	Shifter	7	2	3	21K
	Popper	7	1	1	9.7K
	Filter	7	2	1	12K
	Printer	1	1	1	9.6K

*The size of libDGALS is 123K

The same examples have subsequently been implemented on a heterogeneous mix of underlying physical execution and communication architectures. Table 6.4 shows the different physical implementations used, twelve groups in total. For groups E, F, K, and L two sub-groups are created: machines are distributed on WAN (Internet) and LAN (Intranet). All experimental runs were performed on Intel Core 2 Duo 2.6GHz with 8GB of RAM computation node with Linux 2.6.29.6 as the host OS. 10,000 runs were carried out for each experimental group. Average, median, mode, standard deviation, maximum, and minimum execution times are logged. Average tick count and average tick length of each clock domain are also recorded.

Table 6.4: Varying physical implementation architectures

Experiment group	Clock domain creation	Clock domains created in single or multiple DGALS programs	DGALS programs executed on same or different machines	Effective channel implementation
A	Dynamic	Single	Same	TCP/IP*
B	Dynamic	Single	Same	Shared memory
C	Dynamic	Multiple	Same	TCP/IP
D	Dynamic	Multiple	Same	TCP/IP
E	Dynamic	Multiple	Different	TCP/IP
F	Dynamic	Multiple	Different	TCP/IP
G	Static	Single	Same	TCP/IP*
H	Static	Single	Same	Shared memory
I	Static	Multiple	Same	TCP/IP
J	Static	Multiple	Same	TCP/IP
K	Static	Multiple	Different	TCP/IP
L	Static	Multiple	Different	TCP/IP

*Shared memory is disabled for experiment purpose

6.6.1 The Send-Receive example discussion

Figure 6.10 illustrates the average execution times of groups for the Send-Receive example. The execution time is measured as the time to complete the required computation. The experiments demonstrate the following:

1. The shared-memory based channels perform better than the TCP/IP based counterparts by comparing B to A and H to G, respectively.
2. The static versions of Send-Receive (groups G to L) perform better than the dynamic versions (groups A to F). This could be because dynamic creation introduces overhead, such as handshaking and decoding messages to activate CD.
3. In groups with use of TCP/IP based channels, the execution times are bound by the communication method (underlying network). This applies to both static and dynamic CD creation. For instance, the WAN (Internet) versions of groups E and F are around 16 to 17 times slower than their LAN (Intranet)

counterparts. Similarly, WAN versions of groups K and L perform around 10 times slower than their LAN versions.

4. From groups D, J, F, and L, it can be concluded that when a system is distributed, i.e., implemented as multiple DGALS programs, shared memory is not used in the channel communication.
5. When a DGALS system consists of multiple DGALS programs on the same machine, TCP/IP channels are used. The execution times of such systems (groups C and I) are close to a DGALS system implemented as multiple programs executing on different machines over Intranet (LAN versions of E, F, K, and L). This shows that the overhead of TCP/IP communication plays a significant role and makes a significant contribution to the execution time.

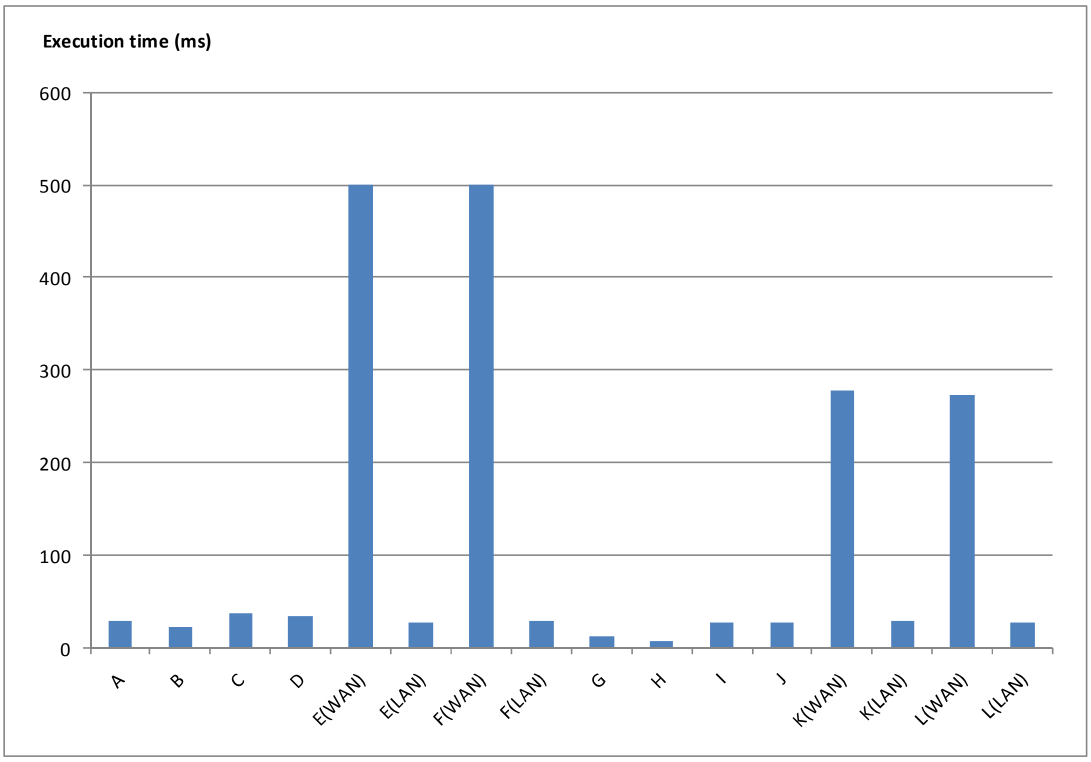


Figure 6.10: Average execution times for Send-Receive example

6.6.2 Discussions of the Sieve example

Average execution times of groups for the Sieve example are illustrated in Figure 6.11. The experiments demonstrate the following:

1. *Shared-memory channels perform faster,* as in the Send-Receive case. 3 to 7 times performance gain is achieved in Sieve compared to 1.33 to 1.76 times of gain in Send-Receive. The difference between Send-Receive and Sieve comes from the fact that the time taken in channel communication in Sieve takes a greater proportion of the overall execution time.
2. *The static versions of Sieve perform worse than the dynamic version*. It was observed that the execution of the static versions utilize the processor (from processor-usage monitors) much more than the dynamic versions. It was also observed that the 'system time' of static versions takes a greater proportion of the execution time than the dynamic ones. This is because only the necessary CDs are active in dynamic sieves. However, in the static version, all 38 CDs are active all the time, consuming significant processor resources, especially with the huge number of very short ticks which occur during channel communication handshaking. This creates a large number of polling-type loops immediately, one after another, resulting in performance degradation. A proposed solution is provided, which adds a short time delay at the end of each tick boundary, explained in detail later.
3. The *bottleneck for dynamic Sieve in terms of execution times is due to the communication medium.* Sieve using LAN (group C) is around 2.5 times faster than the WAN (group E) version. However, the static Sieve, even when implemented over a LAN (group G) connection, does not greatly outperform the WAN-based dynamic Sieve, thus indicating that the performance bottleneck is due to the nature of static CDs, as discussed previously.

As mentioned earlier, when sending and receiving CDs wait for rendezvous over a channel communication, both CDs still carry out logical ticks. Many short ticks will occur in both sending and receiving CDs, in a scenario that both CDs have only one reaction, and are trying to obtain channel rendezvous. This involves continuous checking of the status of the channel as illustrated in Figure 6.12 (a). The overall significance of long ticks decreases due to the huge number of short ticks, hence the reduction of average tick times to unrealistic figures in general.

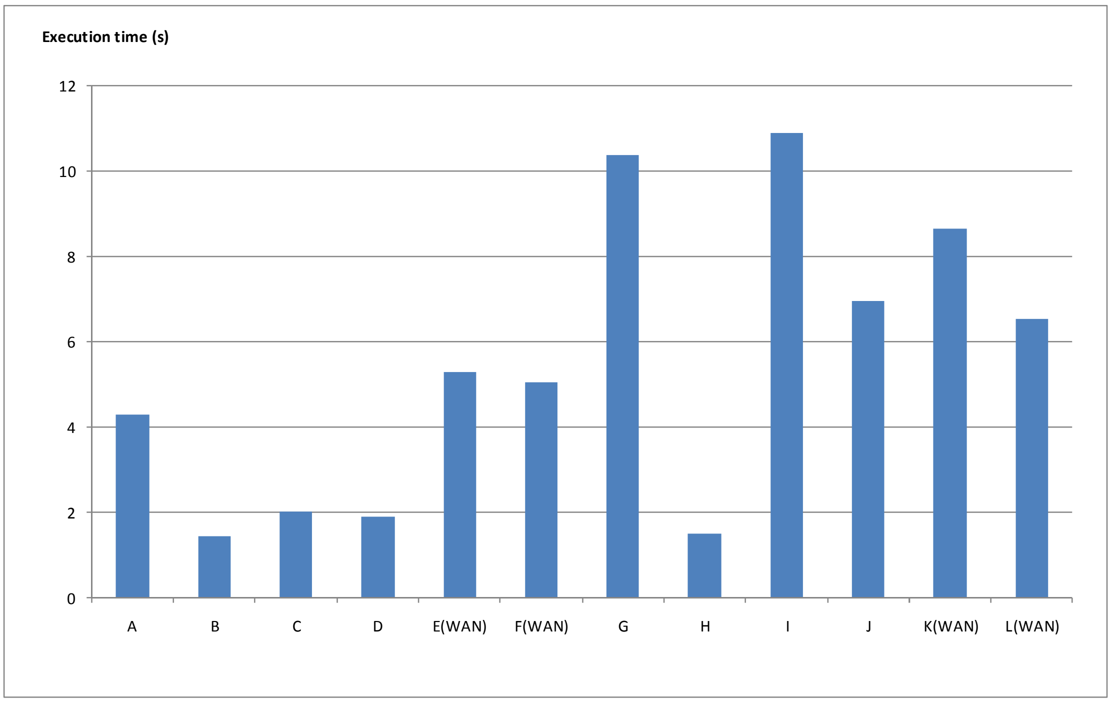


Figure 6.11: Average execution times for Sieve examples

It is possible for short ticks to create polling-like activities similar to a looped behavior with only one pause statement, as illustrated in Figure 6.12 (b). The general solutions to relieving such polling-like executions, which have been also tested, are:

1. *Use signaling such as interrupts and semaphores*: Insert semaphore at the beginning of the input function for each CD, and signal the semaphores from the other thread/interrupt. When the channel is ready, Listener, which governs channel communication, can signal the waiting semaphore in order to continue execution. However, this results in violation of the semantics of ticks, because waiting on a semaphore is a blocking operation that blocks all other synchronous reactions in the same CD, which in turn prevents CDs continuing to carry out any ticks during channel communication. This results in the violation of the GALS semantics (and hence DGALS

semantics) in which other reactions should be able to proceed when one reaction is waiting for rendezvous.

2. *Add time-delay to the loop so that the loop does not iterate as frequently*: This approach preserves the semantics but elongates the tick length (hence the overall execution time) by adding a short time-delay to each tick. The choice of time-delay is important and must be additionally investigated. Choosing a time-delay which is too short will introduce unnecessary overheads due to frequent context switching.

```
While(hand_shaking_conditions)
{

   Pause();
   // checking for channel status
   // which executes in very short time

}
```

(a) A very short tick due to waiting for rendezvous

```
while(condition)
{

   // checking for some variable

}
```

(b) A polling loop in general

Figure 6.12: A very short tick in a while-loop will form a polling-like loop

A one microsecond time-delay in these experiments (this is completely heuristic and, obviously, application dependent) has been chosen. This way, for very short ticks the processor utilization was lowered by 17%. For longer ticks, the delay is only a fraction of the actual computational time and it does not introduce big overheads in timing. The time-delay is added at the beginning of the input function of each clock domain, which is called at each logical tick. Figure 6.13 shows the average execution time for Sieve examples with time-delayed ticks. A performance gain of 9.42 to 46.72 times compared to the non-delayed versions of the Sieves is achieved. For WAN-based groups, performance improves by 1.24 to 1.95 times. CPU utilization is lowered from 90% (the original Sieve implementation) to around 15%, which indicates that a great part of the execution time was due to the short polling-like ticks. System times also decrease drastically. The actual tick lengths are no longer hidden behind the overwhelming number of short ticks. In this case, the average tick duration is not reduced by the meaningless short ticks and hence are close to the realistic and actual execution times of clock domains.

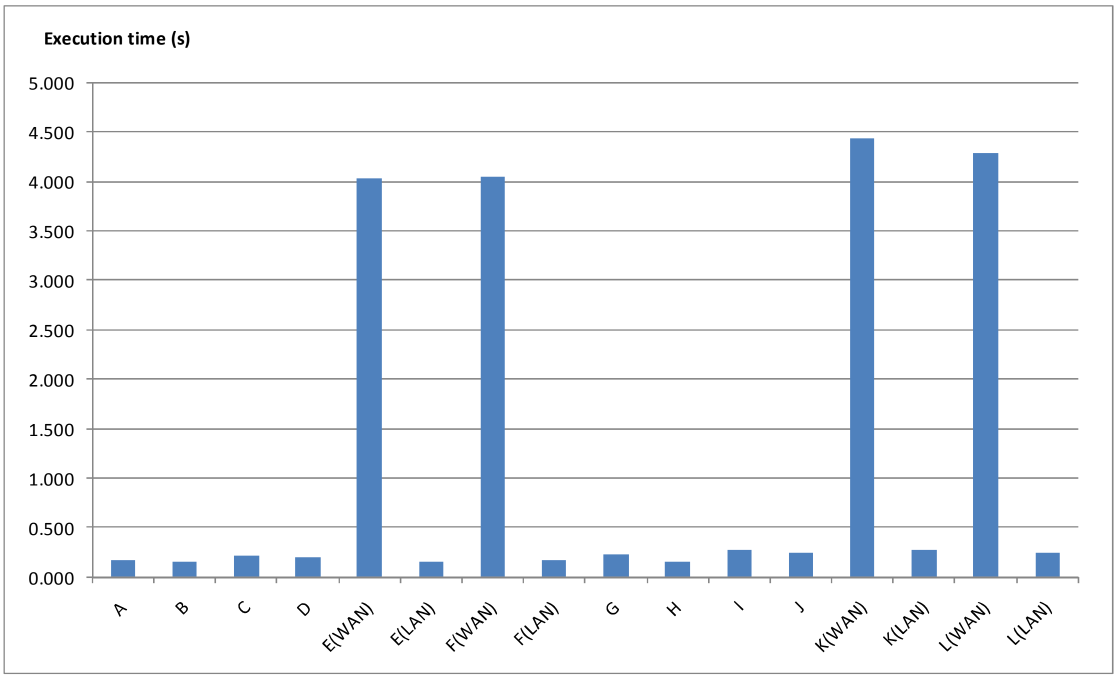


Figure 6.13: Average execution times of Sieves with inserting time-delays in ticks

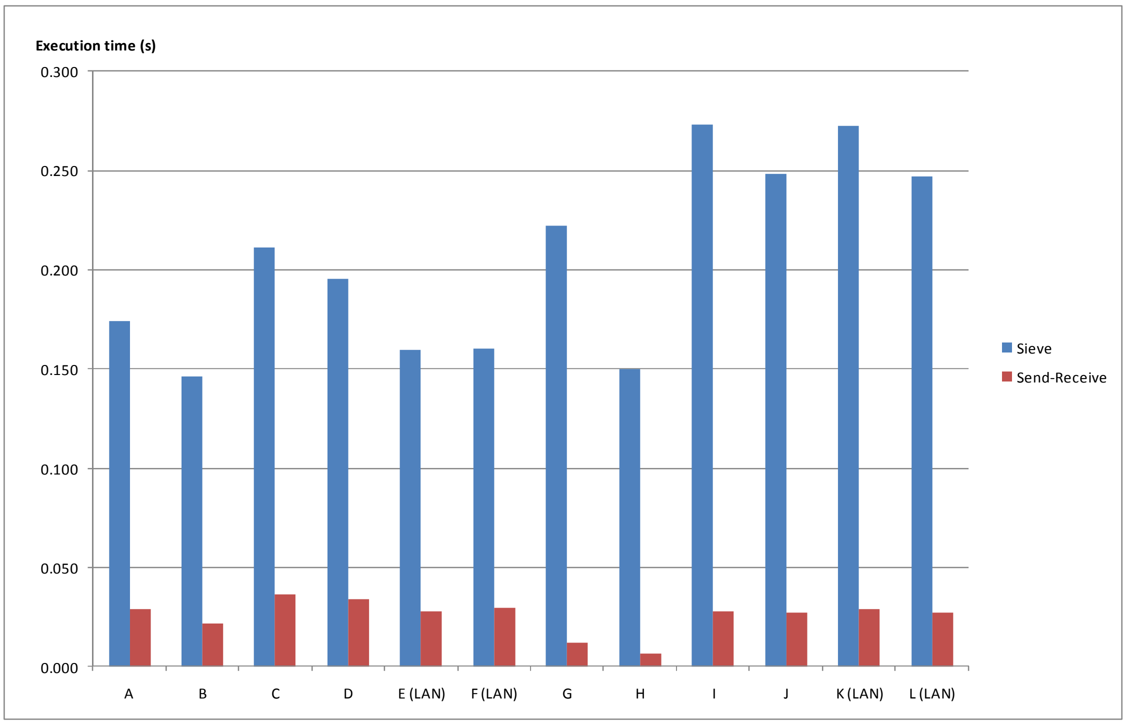


Figure 6.14: Average execution times with time-delayed ticks (without WAN groups)

Figure 6.14 illustrates the execution times of Send-Receive and Sieve shown together for comparison purposes. The WAN-based groups are not included because they have much larger execution times than the other groups because of the slower communication layer and they thus contribute little to the discussion.

In the observations, execution times from Sieve match experimental results of Send-Receive with the following findings:

1. Channels perform better when using shared memory than TCP/IP due to communication overheads with the latter.
2. The dynamic versions outperform static versions when computation (as opposed to communication) forms a significant part of the overall application.
3. Communication medium limits the execution speeds. LAN-based groups have better results than WAN-based.
4. Channels connecting different DGALS programs do not benefit from shared memory regardless of the fact that those programs run on the same computer.
5. From C and I groups, it can be concluded that even though Dynamic GALS programs are located on the same machine, the execution bottleneck remains because of the use of TCP/IP connections.

Experiments show that two features of libDGALS contribute to the improvement of execution times by having: (1) channels implemented with shared memory, and (2) addition of time delay at each tick boundary in order to lower processor utilization. Note that it would also be possible to employ other more efficient communication mechanisms that rely on specific architectural solutions to improve performance of channels, which is a topic of the future work. The static version of Sieve has a smaller memory footprint (150K vs 188.3K) compared to the dynamic version, but does not have the mobility of clock domains, and has a slower overall execution time.

6.6.3 Comparison with other languages and systems

A number of experiments were carried out with different examples and physical execution architecture setups to gauge the effectiveness of our libDGALS approach. Experiments were performed on Linux kernel 2.6.33.3 running on Intel Core 2 Duo 2.6GHz with 4GB RAM. In the distributed scenario, workstations of the same specifications are used. These machines have a slightly better specification than those used for comparison with JADE and DSystemJ reported in [Malik et al., 2010]. The benchmark set is shown in Table 6.5, which also shows the number of lines of source code for each application, together with the memory footprint (generated by the application and one that includes the size of the library). Lines of source code demonstrate the effort required to describe GALS systems and their maintainability. The source code size of DGALS programs is comparable to JADE. With regard to DSystemJ, which is a language-based DGALS approach, the difference varies from 7% for Send-Receive to 92% for Sieve (first 3 columns of Table 6.5). It is worth noting that the structure of each CD plug-in, such as the CDs, channels, signals, and reactions, require explicit definition as compared to DSystemJ, which is a language-based GALS approach, where system structures are abstracted away, hence the smaller source code sizes.

The memory footprint of DGALS programs is comparable to DSystemJ and JADE (3 middle columns of Table 6.5). Because the DGALS library is compact, in contrast to SystemJ and JADE libraries, its programs result in the smallest total memory footprints amongst the three approaches (3 final columns of Table 6.5). Since this approach does not require the JVM, the real memory footprint for DGALS programs is much lower.

Table 6.6 and Table 6.7 present the execution times for three approaches: DSystemJ, JADE, and DGALS. Average tick times are obtained through dividing total execution time by the number of the ticks required to complete the required computations. It is obvious that DGALS programs outperform the functional equivalent models described in DSystemJ and JADE. For the most complex system (security surveillance), the DGALS programs are on average 490 times faster than DSystemJ and 5770 times faster than JADE. Since inter-program CD communication is based on TCP, the communicating DGALS programs on the same machine have similar performance to DGALS programs distributed on LAN.

DGALS programs' smaller memory footprint and much faster execution times through the support of libDGALS makes the DynamicGALS framework more suitable for designing cyber-physical systems compared to both DSystemJ and JADE.

Table 6.5: Lines of code and memory footprint comparisons

Example	Lines of source code			Generated memory footprint (KB)			Total memory footprint (generated + library) (KB)		
	DSystemJ	JADE	libDGALS	DSystemJ	JADE	libDGALS	DSystemJ	JADE	libDGALS
Sieve	163	267	313	99	12	65	216	2623	188
Surveillance system	125	238	216	158	14.5	33.5	265	2625.3	181
Send Receive	39	118	42	38	5.6	18	145	2616.6	141

Table 6.6: Execution time comparisons (Single machine with 2 cores)

Examples	Run-time (ms/tick)													
	DSystemJ					JADE					libDGALS			
Send-Receive	CD1	CD2				CD1		CD2			CD1	CD2		
	5	5.57				74.7		185.9			0.009	0.008		
Sieve	CD1	CD2	CD3	CD4	CD5	CD1	CD2	CD3	CD4	CD5	CD1	CD2	CD3	CD4
	0.1	17	16.7	23.4	17	1	340	361.8	322.4	514	0.25	0.33	1.31	0.74

Table 6.7: Execution time comparisons (Distributed system 2 machines 4 cores each)

Example	Run-time (ms/tick)											
	DSystemJ				JADE				libDGALS			
Send Receive	CD1		CD2		CD1		CD2		CD1		CD2	
	20.7		22.2		86.88		470		0.009		0.01	
Surveillance system	CD1	CD2	CD3	CD4	CD1	CD2	CD3	CD4	CD1	CD2	CD3	CD4
	202.7	191.4	125.1	133.7	3243.4	1498.1	1320.6	1603	0.418	0.457	0.253	0.217

6.7 Summary

This chapter describes the DynamicGALS framework designed to support programming of dynamic systems based on the formal Globally Asynchronous Locally Synchronous (GALS) Model of Computation (MoC). The DynamicGALS framework enables programmers to describe simple to large scale DGALS systems by using CD plug-ins. Dynamic creation of CDs and channels in the CD plug-in instances, along with 'weak' CD mobility, are provided in an API to strengthen the design capability, thereby making the DynamicGALS framework suitable for implementing a wide range of dynamic distributed systems. The framework also provides an abstract means of programming reactivity and composition of synchronous concurrent processes using behavioral hierarchy. The approach separates the design and modeling of the system from the underlying physical execution and communication layer. This allows changing model and physical layers independently without affecting each other. The DynamicGALS framework allows the utilizing of a mixture of different execution and communication architectures with ease and efficiency. Being based on C, it allows easy integration of legacy code. Future work includes graphical tools for describing DGALS programs and systems to reduce design effort, as well as building tools for automated mapping of CDs and DGALS programs to heterogeneous architectures that will enable creation of virtual topologies.

7

Conclusions and future work

With increasing complexity in system design, adopting a higher level of abstraction and applying design with formal models of computation reduces design effort and ensures the correctness of the design. Several approaches that enhance system design have been proposed and developed and are detailed in Chapter 2. Chapters 3 to 6 detail the development of a library-based approach to support both GALS and DGALS MoC system design from programming language C, which is still a major language in embedded systems design. In this chapter, a summary of the work presented in this thesis, as well as its conclusions, is given, along with plans for possible future works.

7.1 Conclusions

Discussions of system level-design are detailed in Chapter 3. System-level design, which can be categorized into system-level synthesis, component-based design, and platform-based design, are performed according to the available resources and knowledge of the target platform. The design that can be further divided into stages consists of the specification model, the functional model, component-emergence model, bus-architecture model, behavior model, bus-functional model, cycle-accurate model, and implementation model. Software and hardware partitioning are realized throughout the refinements. Operating systems, used to manage software concurrency, serve as the bridge between software and hardware of the systems, and play an important role in the design of systems that include software-implemented functionalities. The modeling of operating systems is of interest in order to achieve a model of the whole software of the designed system and can be carried out in different granularities of accuracy. In Chapter 3, an OS model has been developed in SYSTEMC. The model provides a number of services which can be used by application processes. Signal-operation services, which are described as part of the core services in the OS model, are used to support reactive behaviors which can be specified in synchronous/reactive language such as ESTEREL. Case studies have been implemented to justify the necessities of having signal-operation services for implementing reactive systems with conventional OS services. The implementation of signal-operation services can be built-in as part of a kernel or as a user-level library. The concept of signal-operation services is further extended and developed resulting in a library-based approach, libGALS, detailed in Chapter 4.

libGALS provides a more powerful mechanism and allows both synchronous and asynchronous concurrency to be incorporated as a single, correct libGALS program, that complies with the globally asynchronous locally synchronous (GALS) MoC. Within a libGALS program, the overall behaviors of system are first divided into groups of asynchronous clock domains. Finer grain concurrency in each clock domain is implemented in the form of synchronous reactions, which within the same clock domains, are executed in logical time steps called ticks, being the same as systems described in synchronous languages. Synchronous reactions of the same clock domain communicate with each other through signal broadcasting. On the other hand, reactions of different clock domains send and receive information to and from each other through the use of channels which in libGALS programs follow the semantics of CSP rendezvous. libGALS is implemented based on primitive services provided by the operating systems, such as thread creations and semaphores. Each synchronous reaction is mapped to a thread of the libGALS program, and is governed by the programmer invisible thread: Synchronizer. Each clock domain is equipped with one Synchronizer which uses semaphores, provided by the OS to resolve dependencies between reactions, as locks. Because of the thread-based approach, libGALS can benefit from the multicore/multiprocessor architecture. The libGALS approach is the first known library-based approach that supports programming GALS systems. SystemJ, a language-based approach, is compared with libGALS in Chapter 4.

libGALS enables designers to construct correct-by-design software programs given that the software is described correctly with regard to the specification. Behavior of programs can be also seen as behaviors of the underlying processor(s) in the system model. To present a system model with correct programs, a framework for integrating libGALS programs into the SYSTEMC modeling environment, called GALS-Designer, has been developed, as detailed in Chapter 5. libGALS programs are wrapped to SYSTEMC modules through the use of macros and static functions in C++. Because libGALS makes use of the multicore/multiprocessor of the simulation host, the simulation speed of libGALS-SYSTEMC modules is greatly enhanced. Therefore the GALS-Designer framework provides feasibilities for both describing correct software programs and fast simulation speed. Furthermore, GALS-Designer also enables the exploration of distributing GALS systems into single or multiple libGALS programs. The latter can be mapped into different processors locally (on the same platform) or different machines on distributed platforms. Communication between libGALS programs in GALS-Designer is achieved through the help of SYSTEMC modeling techniques.

To further explore dynamicity in distributed systems, enhancements such as creating clock domains in run-time on different computational nodes have been added to libGALS, resulting in a library called libDGALS, which follows the Dynamic GALS (DGALS) MoC. The DynamicGALS framework, which provides both interfaces to program libDGALS programs and run-time support for them, is detailed in Chapter 6. Each libDGALS program is similar to a libGALS program, and hosts a number of clock domains. libDGALS program is further equipped with specialized Listener threads to handle clock domain creations and communications between libDGALS programs. The libDGALS library, inherited from its predecessor libGALS, requires minimal support from the OS. In this case, thread creation, semaphore, and networking stack are the only requirements on the underlying OS. The code size and high performance of libDGALS programs are compared with the language-based counter-parts, DSystemJ at the end of the Chapter 6.

7.2 Future research

7.2.1 Hardware support for libGALS and libDGALS

Based on the results from OS modeling and simulations, possible and preferred configuration (HW/SW partitioning) for the OS implementation can be obtained as shown in Chapter 3. Hardware support to the OS thus needs further investigation to support GALS MoC. The current functional unit to support reactivity is available through customization of processors but does not have the support from the OS which is required for libGALS and libDGALS. Such support can be similar to RTM proposed in [Kohout et al., 2004] by applying dependency resolution in the scheduling policy which operates in hardware.

7.2.2 Exploration of styles of concurrent execution

Future work will explore how to manage and achieve even higher performance gains by controlling processor affinity of libGALS and libDGALS. The scheduling of synchronous reactions is governed by the underlying scheduling policy of the operating systems. The operating systems generally follow either priority-based scheduling or fair-for-all scheduling. Priority-based scheduling is not used by synchronous reactions in the same clock domain, because it is not necessary; Synchronizer will handle the execution sequences by resolving the dependencies, executed as the lowest priority process. Fair scheduling is often adopted by general operating systems also, as in implementation of libGALS and libDGALS on these systems. However, in control-dominated applications, performance of both libGALS and libDGALS programs might suffer from unnecessary ticks while performing communications between clock domains. In this case, controlling processor affinity to achieve best execution performance of clock domains will be investigated. For example, a clock domain may be suspended when it is only waiting for the rendezvous on channel communication to prevent unnecessary ticks elapse, and thus lower the performance of the overall system. Such an approach can be adopted in GALS-Designer, which also relies on the execution model of libGALS, in order to increase the simulation speed.

Furthermore, libDGALS is currently built with weak mobility, that is, new instances of clock domains are created without ‘previous memories’ (previous working state of the clock domain), unless giving all the required information as the argument upon activation of the clock domain. Investigation to include the thread/process state of each reaction to enable strong mobility will be carried out as future work.

7.2.3 Designer-friendly framework

Glue-logic such as SyncNodes in GALS-Designer is used to integrate libGALS programs to libGALS-SYSTEMC modules. This glue-logic is currently presented in the form of source codes, which are prone to programmers’ errors, such as accidental modification of the source code. In order to resolve this issue, parsers of a libGALS program can be used to generate essential parts of libGALS-SYSTEMC module, by checking clock domains and reactions in the program sources. On top of this approach, a GUI will be developed as a part of GALS-Designer. It will reduce the amount of textual information entered by the designers to prevent programming errors. The GUI will automatically generate templates of libGALS programs, and the designer will only need to populate algorithmic parts.

An approach of using animation tools to model complex dynamic systems has been introduced in [Efroni et al., 2005]. GALS systems involve execution flows of clock domains, communications between clock domains, and dynamic creations of clock domains, which can be presented in a similar manner. Through the use of animation tools, along with the other graphical tools, specification of GALS systems and activities within can be modeled and observed in an intuitive fashion. This would also prevent manual coding which leads to programmatic error due to the human factor, such as incorrect channel creations, i.e. sending and receiving clock domains are invalid.

The GUI approaches can also be applied to the DynamicGALS framework. For example, the designer should be able to see the initial state of the DGALS systems, such as the available storage and resources of each computational node, to estimate if a clock domain can be spawned and perform correctly on the target computational node. Also, the default configuration should also be generated automatically to prevent programmers' errors.

7.2.4 Better support for embedded systems

Overheads may occur when designing systems with very fine grain concurrency, as for instance, having many concurrent synchronous reactions with very tiny numbers of operations to perform. In this case, synchronization overheads may annul the actual performance gain from the multicore systems, because of context switching. There are approaches to prevent heavy context switching (or no context switching is required) on the operating systems level, that are adopted in researches of sensor networks. For instance, the operating system Contiki applies the uses of protothread [Dunkels et al., 2006] to be executed on platforms that require low memory footprint.

With tight merging of minimal functionalities of the operating system (particularly scheduling and support for dynamic loading) with a library-based approach, an operating system might not be required anymore. This would be suitable on bare-bone processors. This approach would place an abstract machine above the hardware of traditional processors, and would equally support language-based systems (compilers) and library-based systems as libDGALS.

This research should result in an abstract machine ready to use for implementation of libDGALS programs on distributed platforms that include wireless sensor networks based on more powerful processors (e.g. ARM-type) and also open a research line resulting in specification of desired features of the processors that would directly support DynamicGALS MoC.

7.2.5 Provide mappings of existing MoCs

Applying a library- and language- based approach to model process network has been proposed in some earlier works. For example, NRP (nets of reactive processes) [Boussinot, 1992] implements KPN-like systems with SUGRARCUBES (as an extension based on Java). Similarly, Synchronous Kahn Network [Caspi & Pouzet, 1996] is proposed with programming in the style of functional language, which relies on the support of the dedicated compiler. Synchronous Kahn Network can be seen as one of the solutions to KPN by applying the concepts of synchrony.

Similarly, investigation should be carried out to map a PN/KPN-like approach to libGALS, or even with distributed support as presented in libDGALS. A fix-rated-based approach to KPN, such as SDF, can also be adopted while the rate can be computed on-run-time as part of the investigation into scheduling policy mentioned in Section 7.2.2.

7.2.6 Support of verification

As a library-based approach, libGALS and libDGALS leave to the designers some of the responsibility of constructing correct programs. It is thus possible for a designer to write a compliant program, which, while not violating the syntax of the base language (i.e. C), does behave incorrectly. This problem does not exist in a language-based approach such as SystemJ and DSystemJ. Because libGALS and libDGALS share similar features to SystemJ and DSystemJ, it is possible to extract, or to map the control part of the language, to both SystemJ and DSystemJ or other similar languages to perform a static check, e.g. verification. Static checking on programs also opens doors to other verification methodologies mentioned in the dynamic languages such as DSL presented in [Attar et al., 2011].

References

[Acosta-Bermejo, 1999] R. Acosta-Bermejo, “Programming in REJO,” 1999

[Acosta-Bermejo, 2000] R. Acosta-Bermejo, “Reactive operating system, reactive java objects,” *Proc. NOTERE’2000, ENST, Paris*, 2000

[Adomat et al., 2002] J. Adomat, J. Furunas, L. Lindh, and J. Starner, “Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems,” *Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on*, 2002, pp. 164–168

[Agha, 1985] G. A. Agha, “Actors: a model of concurrent computation in distributed systems,” 1985

[André, 1995] C. André, “SyncCharts: A visual representation of reactive behaviors,” *Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis*, 1995

[André, 1996] C. André, “Representation and analysis of reactive behaviors: A synchronous approach,” 1996

[André, 2003] C. André, *Semantics of syncharts*, Laboratoire I3S - Sophia Antipolis, 2003

[Andre & Péraldi, 1993] C. Andre and M. A. Péraldi, “Effective implementation of ESTEREL programs,” *Real-Time Systems, 1993. Proceedings., Fifth Euromicro Workshop on*, 1993, pp. 262–267

[Antonotti et al., 2000] M. Antonotti, A. Ferrari, A. Flesca, and A. Sangiovanni-Vincentelli, “JESTER: An Esterel based reactive Java extension for reactive embedded systems,” *Forum on specification & Design Languages*, 2000

[Armstrong et al., 1993] J. Armstrong, R. Virding, C. Wikström, and M. Williams, “Concurrent programming in ERLANG,” 1993

[Arnold et al., 2000] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*, Addison-Wesley Reading, MA, 2000

[Astley, 1999] M. Astley, *The Actor Foundry. University of Illinois*, 1999

[Attar et al., 2011] P. Attar, F. Boussinot, L. Mandel, and J. F. Susini, “Proposal for a Dynamic Synchronous Language,” 2011

[Balarin et al., 1997] F. Balarin, P. Di Giusto, A. Jurecska, M. Chiodo, C. Passerone, H. Hsieh, A. Sangiovanni-Vincentelli, E. Sentovich, B. Tabbara, L. Lavagno, and others, *Hardware-software co-design of embedded systems: the POLIS approach*, Springer Netherlands, 1997

[Barnes, 2005] F. Barnes, “Interfacing C and occam-pi,” 2005

[Barry, 2008] R. Barry, “FreeRTOS,” *Internet, Oct*, 2008

[Bellifemine et al., 2005] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, “JADE—a java agent development framework,” *Multi-Agent Programming*, 2005, pp. 125–147

[Benveniste et al., 1985] A. Benveniste, P. Bournai, T. Gautier, and P. Le Guernic, “SIGNAL: a data flow oriented language for signal processing,” 1985

[Benveniste et al., 2003] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, 2003, pp. 64–83

[Benveniste & Berry, 1991] A. Benveniste and G. Berry, “The synchronous approach to reactive and real-time systems,” *Proceedings of the IEEE*, vol. 79, 1991, pp. 1270–1282

[Berry et al., 1983] G. Berry, S. Moisan, and J. P. Rigault, “Esterel: Towards a synchronous and semantically sound high-level language for real-time applications,” *Proc. IEEE Real-Time Systems Symposium*, 1983, pp. 30–40

[Berry, 1992] G. Berry, “A hardware implementation of pure Esterel,” *Sadhana*, vol. 17, 1992, pp. 95–130

[Berry et al., 1993] G. Berry, S. Ramesh, and R. K. Shyamasundar, “Communicating reactive processes,” *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 85–98

[Berry, 1993] G. Berry, “The semantics of pure Esterel,” *Program Design Calculi*, vol. 118, 1993, pp. 361–409

[Berry, 1999] G. Berry, “The constructive semantics of pure Esterel,” 1999

[Berry, 2000] G. Berry, *The Esterel v5 language primer. Version 5 91 (2000)*, 2000

[Berry, 2004] G. Berry, “Programming and Verifying an Elevator in Esterel v7,” *Esterel Technologies*, 2004

[Berry & Cosserat, 1984] G. Berry and L. Cosserat, “The synchronous programming language Esterel and its mathematical semantics,” *Seminar on Concurrency*, 1984, pp. 389–448

[Berry & Gonthier, 1988] G. Berry and G. Gonthier, “The ESTEREL synchronous programming language : design, semantics, implementation,” 1988

[Berry & Sentovich, 2000] G. Berry and E. M. Sentovich, “An implementation of constructive synchronous programs in POLIS,” *Formal Methods in System Design*, vol. 17, 2000, pp. 135–161

[Black et al., 2008] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: from the ground up*, Springer Verlag, 2008

[Boniol & Adelantado, 1993] F. Boniol and M. Adelantado, “Programming Communicating Distributed Reactive Automata: the Weak Synchronous Paradigm,” 1993

[Bouchhima et al., 2004] A. Bouchhima, S. Yoo, and A. Jeraya, “Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model,” *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, 2004, pp. 469–474

[Boudol, 2004] G. Boudol, “ULM: A core programming model for global computing,” *Programming Languages and Systems*, 2004, pp. 234–248

[Boussinot, 1991] F. Boussinot, “Reactive C: An extension of C to program reactive systems,” *Software: Practice and Experience*, vol. 21, Apr. 1991, pp. 401–428

[Boussinot, 1992] F. Boussinot, “Reseaux de processus reactifs,” 1992

[Boussinot et al., 1996] F. Boussinot, G. Doumenc, and J. B. Stefani, “Reactive objects,” *Annals of Telecommunications*, vol. 51, 1996, pp. 459–473

[Boussinot, 1996] F. Boussinot, *Icobj programming*, Citeseer, 1996

[Boussinot, 1996] F. Boussinot, *La programmation réactive : Application aux systèmes communicants*, Masson, 1996

[Boussinot et al., 1999] F. Boussinot, L. Hazard, and J. F. Susini, “The junior reactive kernel,” *INRIA Research Report*, vol. 3732, 1999

[Boussinot, 2002] F. Boussinot, “Fair Threads in C,” 2002

[Boussinot, 2005] F. Boussinot, “Loft+Cyclone,” 2005

[Boussinot & Dabrowski, 2006] F. Boussinot and F. Dabrowski, “Cooperative Threads and Preemptive Computations,” 2006

[Boussinot & Dabrowski, 2007] F. Boussinot and F. Dabrowski, “Safe reactive programming: The FunLoft proposal,” 2007

[Boussinot & Hazard, 1996] F. Boussinot and L. Hazard, “Reactive scripts,” *rtcsa*, 1996, p. 270

[Boussinot & De Simone, 1996] F. Boussinot and R. De Simone, “The SL synchronous language,” *Software Engineering, IEEE Transactions on*, vol. 22, 1996, pp. 256–266

[Boussinot & Susini, 1997] F. Boussinot and J.-F. Susini, “The SugarCubes Tool Box,” Sep. 1997

[Boussinot & Susini, 1998] F. Boussinot and J. F. Susini, “The SugarCubes tool box: a reactive Java framework,” *Software: Practice and Experience*, vol. 28, 1998, pp. 1531–1550

[Bovet et al., 2002] D. Bovet, M. Cesati, and A. Oram, *Understanding the Linux kernel*, O’Reilly & Associates, Inc. Sebastopol, CA, USA, 2002

[Brandt & Schneider, 2008] J. Brandt and K. Schneider, “How different are Esterel and SystemC,” *Embedded Systems Specification and Design Languages*, 2008, pp. 3–13

[Browne & Clarke, 1985] M. C. Browne and E. M. Clarke, *SML - a high level language for the design and verification of finite state machines*, Department of Computer Science, Carnegie-Mellon University, 1985

[Cai et al., 2003] L. Cai, L. Cai, S. Verma, S. Verma, D. D. Gajski, and D. D. Gajski, *Comparison of SpecC and SystemC Languages for System Design*, 2003

[Cai & Gajski, 2003] L. Cai and D. Gajski, “Transaction level modeling: an overview,” *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2003, p. 24

[Caspi et al., 1987] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: a declarative language for real-time programming,” *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 178–188

[Caspi et al., 2007] P. Caspi, G. Hamon, and M. Pouzet, “Synchronous Functional Programming with Lucid Synchrone,” 2007

[Caspi & Girault, 1995] P. Caspi and A. Girault, “Execution of distributed reactive systems,” *EURO-PAR ’95 Parallel Processing*, S. Haridi, K. Ali, and P. Magnusson, Eds., Berlin/Heidelberg: Springer-Verlag, 1995, pp. 13–26

[Caspi & Pouzet, 1996] P. Caspi and M. Pouzet, “Synchronous kahn networks,” *ACM SIGPLAN Notices*, 1996, pp. 226–238

[Cesario et al., 2002] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, “Component-based design approach for multicore SoCs,” *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 789–794

[Cesário et al., 2002] W. O. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, L. Gauthier, and M. Diaz-Nava, “Multiprocessor SoC platforms: a component-based design approach,” *Design & Test of Computers, IEEE*, vol. 19, 2002, pp. 52–63

[Chapiro, 1984] D. M. Chapiro, “Globally-asynchronous locally-synchronous systems,” PhD Thesis, Stanford University, 1984

[Charles et al., 2005] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, San Diego, CA, USA: ACM, 2005, pp. 519–538

[Cheong et al., 2003] E. Cheong, J. Liebman, J. Liu, and F. Zhao, “TinyGALS: A programming model for event-driven embedded systems,” *Proceedings of the 2003 ACM symposium on Applied computing*, 2003, pp. 698–704

[Chevalier et al., 2006] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, and G. Bois, “A SystemC refinement methodology for embedded software,” *Design & Test of Computers, IEEE*, vol. 23, 2006, pp. 148–158

[Church, 1932] A. Church, “A set of postulates for the foundation of logic,” *The Annals of Mathematics*, vol. 33, 1932, pp. 346–366

[Clarke, 1997] E. Clarke, “Model checking,” *Foundations of Software Technology and Theoretical Computer Science*, 1997, pp. 54–56

[Clarke Jr et al., 1991] E. M. Clarke Jr, D. E. Long, and K. L. McMILLAN, “A language for compositional specification and verification of finite state hardware controllers,” *Proceedings of the IEEE*, vol. 79, 1991, pp. 1283–1292

[Clinger, 1981] W. D. Clinger, “Foundations of actor semantics,” 1981

[Closse et al., 2002] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, “SAXO-RT: Interpreting Esterel semantic on a sequential execution structure,” *Electronic Notes in Theoretical Computer Science*, vol. 65, 2002, pp. 80–94

[Colby et al., 1998] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Laufer, and C. Puchol, “Design and implementation of Triveni: a process-algebraic API for threads+ events,” *Computer Languages, 1998. Proceedings. 1998 International Conference on*, 1998, pp. 58–67

[Connell, 2003] J. Connell, “ARM System-Level Modeling,” *White paper, June*, vol. 25, 2003

[Dagum & Menon, 2002] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, 2002, pp. 46–55

[Dayaratne, 2004] M. W. . Dayaratne, “Direct execution of Esterel using reactive microprocessors,” Electrical and Computer Engineering)–University of Auckland, 2004

[Desmet et al., 2000] D. Desmet, D. Verkest, and H. De Man, “Operating system based software generation for systems-on-chip,” *Proceedings of the 37th Annual Design Automation Conference*, 2000, p. 401

[Doumenc & Boussinot, 1991] G. Doumenc and F. Boussinot, “La Programmation par Objets Reactifs (POR),” *Rapport Interne ENSMP-CMA*, 1991

[Dunkels et al., 2006] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “Protothreads: Simplifying event-driven programming of memory-constrained embedded systems,” *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 29–42

[Edwards, 2002] S. A. Edwards, “An Esterel compiler for large control-dominated systems,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, 2002, pp. 169–183

[Edwards, 2003] S. A. Edwards, “Making cyclic circuits acyclic,” *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 159–162

[Edwards et al., 2006] S. A. Edwards, V. Kapadia, and M. Halas, “Compiling Esterel into Static Discrete-Event Code,” *Electronic Notes in Theoretical Computer Science*, vol. 153, Jun. 2006, pp. 117–131

[Edwards & Tardieu, 2006] S. A. Edwards and O. Tardieu, “SHIM: A deterministic model for heterogeneous embedded systems,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, 2006, pp. 854–867

[Edwards & Zeng, 2007] S. A. Edwards and J. Zeng, “Code generation in the Columbia Esterel compiler,” *EURASIP Journal on Embedded Systems*, vol. 2007, 2007, pp. 1–31

[Efroni et al., 2005] S. Efroni, D. Harel, and I. R. Cohen, “Reactive animation: Realistic modeling of complex dynamic systems,” *Computer*, vol. 38, 2005, pp. 38–47

[Engelschall & Pth, 2006] R. S. Engelschall and G. Pth, *Gnu portable threads*, June, 2006

[Formaggio et al., 2004] L. Formaggio, F. Fummi, and G. Pravadelli, “A timing-accurate HW/SW co-simulation of an ISS with SystemC,” *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 152–157

[Fuggetta et al., 1998] A. Fuggetta, G. P. Picco, and G. Vigna, “Understanding code mobility,” *Software Engineering, IEEE Transactions on*, vol. 24, 1998, pp. 342–361

[Gädtke et al., 2007] S. Gädtke, C. Traulsen, and R. von Hanxleden, “HW/SW co-design for Esterel processing,” *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 99–104

[Gajski et al., 2000] D. D. Gajski, J. Zhu, R. D\ömer, A. Gerstlauer, and S. Zhao, *SpecC: specification language and methodology*, Springer Netherlands, 2000

[Galletly, 1990] J. Galletly, *Occam 2*, Taylor & Francis, 1990

[Gauthier et al., 2002] L. Gauthier, S. Yoo, and A. A. Jerraya, “Automatic generation and targeting of application-specific operating systems and embedded systems software,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, 2002, pp. 1293–1301

[Gerstlauer et al., 2003] A. Gerstlauer, H. Yu, and D. D. Gajski, “RTOS modeling for system level design,” *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 130–135

[Gery et al., 2002] E. Gery, D. Harel, and E. Palachi, “Rhapsody: A complete life-cycle model-based development system,” *Integrated Formal Methods*, 2002, pp. 1–10

[Ghenassia, 2005] F. Ghenassia, *Transaction-level modeling with Systemc: TLM concepts and applications for embedded systems*, Springer Verlag, 2005

[Girault, 2005] A. Girault, “A survey of automatic distribution method for synchronous programs,” *International workshop on synchronous languages, applications and programs, SLAP*, 2005

[Gropp et al., 1999] W. Gropp, E. Lusk, and A. Skjellum, “Using MPI: portable parallel programming with the message passing interface,” 1999

[Grötker et al., 2002] T. Grötker, S. Liao, G. Martin, and S. Swan, *System design with SystemC*, Springer Netherlands, 2002

[Gruian et al., 2006] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, “The SystemJ approach to system-level design,” *Formal Methods and Models for Co-Design, 2006. MEMOCODE ’06. Proceedings. Fourth ACM and IEEE International Conference on*, 2006, pp. 149–158

[Le Guernic et al., 1991] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, “Programming real-time applications with SIGNAL,” *Proceedings of the IEEE*, vol. 79, 1991, pp. 1321–1336

[Le Guernic et al., 2003] P. Le Guernic, J. P. Talpin, and J. C. Le Lann, “Polychrony for system design,” *JOURNAL OF CIRCUITS SYSTEMS AND COMPUTERS*, vol. 12, 2003, pp. 261–304

[Halbwachs et al., 1986] N. Halbwachs, A. Lonchampt, and D. Pilaud, “Describing and designing circuits by means of a synchronous declarative language,” *From HDL Descriptions to Garanteed Correct Circuits Designs*, 1986, pp. 255–268

[Halbwachs et al., 1991] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data flow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, 1991, pp. 1305–1320

[Halbwachs, 1998] N. Halbwachs, “Synchronous programming of reactive systems,” *Computer Aided Verification*, 1998, pp. 1–16

[Haller & Odersky, 2009] P. Haller and M. Odersky, “Scala Actors: Unifying thread-based and event-based programming,” *Theoretical Computer Science*, vol. 410, Feb. 2009, pp. 202–220 [von Hanxleden, 2009] R. von Hanxleden, “SyncCharts in C - A Proposal for Light-Weight Deterministic Concurrency,” *ACM Embedded Software Conference (EMSOFT)*, 2009, pp. 11–16

[Harel, 1987] D. Harel, “Statecharts: a visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, Jun. 1987, pp. 231–274

[Harel & Pneuli, 1985] D. Harel and A. Pneuli, *On the development of reactive systems*, Microelectronics and Computer, Technology Corporation, 1985

[Haverinen et al., 2002] A. Haverinen, M. Leclercq, N. Weyrich, and D. Wingard, “SystemC based SoC communication modeling for the OCP protocol,” *Whitepaper, October*, 2002

[Hawkins, 2011] T. Hawkins, “Atom,” 2011

[Hazard et al., 1999] L. Hazard, J.-F. Susini, and F. Boussinot, “The Junior Reactive Kernel,” Jul. 1999

[Herrera et al., 2003] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, “Systemic embedded software generation from systemC,” *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, 2003, p. 10142

[Hewitt et al., 1973] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” *Proceedings of the 3rd international joint conference on Artificial intelligence*, 1973, pp. 235–245

[Hilderink et al., 1999] G. Hilderink, J. Broenink, A. Bakkers, and N. C. Schaller, “Communicating Threads for Java^{TM},” *Architectures, languages and techniques for concurrent systems: WoTUG-22, proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, 11-14 April 1999, Keele, United Kingdom*, 1999, p. 243

[Hoare, 1978] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, 1978, pp. 666–677

[Honda & Takada, 2003] S. Honda and H. Takada, “Evaluation of applying SpecC to the integrated design method of device driver and device,” 2003

[IEEE, 2000] IEEE, “IEEE Standard VHDL Language Reference Manual,” 2000

[IEEE, 2001] IEEE, “IEEE Standard Verilog Hardware Description Language,” 2001

[IEEE, 2008] IEEE, “IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7,” *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, 2008

[Jones, 2003] S. P. Jones, *Haskell 98 language and libraries: the revised report*, Cambridge Univ Pr, 2003

[Jose et al., 2009] B. A. Jose, H. D. Patel, S. K. Shukla, and J. P. Talpin, “Generating multi-threaded code from polychronous specifications,” *Electronic Notes in Theoretical Computer Science*, vol. 238, 2009, pp. 57–69

[Kahn, 1974] G. Kahn, “The semantics of a simple language for parallel programming,” 1974

[Kohout et al., 2004] P. Kohout, B. Ganesh, and B. Jacob, “Hardware support for real-time operating systems,” *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, 2004, pp. 45–51

[Krause et al., 2008] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel, “Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation,” *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2008, pp. 143–148

[Krogh et al., 2008] B. Krogh, E. Lee, I. Lee, A. Mok, R. Rajkumar, L. Sha, A. Vincentelli, K. Shin, J. Stankovic, J. Sztipanovits, and others, “Cyber-Physical Systems, Executive Summary,” *Cyber-Physical Systems Summit*, 2008

[Krstić et al., 2007] M. Krstić, E. Grass, F. K. G\ürkaynak, and P. Vivet, “Globally asynchronous, locally synchronous circuits: Overview and outlook,” *IEEE Design and Test*, 2007, pp. 430–441

[Labrosse, 2002] J. J. Labrosse, *MicroC/OS-II: the real-time kernel*, Newnes, 2002

[Lavagno & Sentovich, 1999] L. Lavagno and E. Sentovich, “ECL: A specification environment for system-level design,” *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 511–516

[Lavender & Schmidt, 1995] R. G. Lavender and D. C. Schmidt, “Active object–an object behavioral pattern for concurrent programming,” 1995

[Lee et al., 2003] J. Lee, V. J. Mooney III, A. Daleby, K. Ingstr\öm, T. Klevin, and L. Lindh, “A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS,” *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, 2003, pp. 683–688

[Lee, 2006] E. A. Lee, “The problem with threads,” *Computer*, vol. 39, 2006, pp. 33–42

[Lee & Messerschmitt, 1987] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *Computers, IEEE Transactions on*, vol. 100, 1987, pp. 24–35

[Lee & Neuendorffer, 2005] E. A. Lee and S. Neuendorffer, “Concurrent models of computation for embedded software,” *Computers and Digital Techniques, IEE Proceedings-*, 2005, pp. 239–250

[Lee & Sangiovanni-Vincentelli, 1998] E. A. Lee and A. Sangiovanni-Vincentelli, “A framework for comparing models of computation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, 1998, pp. 1217–1229

[Li et al., 2006] X. Li, M. Boldt, and R. von Hanxleden, “Mapping Esterel onto a multi-threaded embedded processor,” *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 303–314

[Lipsett et al., 1986] R. Lipsett, E. Marschner, and M. Shahdad, “VHDL-The language,” *Design & Test of Computers, IEEE*, vol. 3, 1986, pp. 28–41

[Liskov & Shrira, 1988] B. Liskov and L. Shrira, *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, ACM, 1988

[Madsen et al., 2004] J. Madsen, K. Virk, and M. Gonzales, “Abstract RTOS modeling for multiprocessor system-on-chip,” *System-on-Chip, 2003. Proceedings. International Symposium on*, 2004, pp. 147–150

[Malik, 1994] S. Malik, “Analysis of cyclic combinational circuits,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, 1994, pp. 950–956

[Malik et al., 2010] A. Malik, A. Girault, and Z. Salcic, “The DSystemJ programming language for dynamic GALS systems: it’s semantics, compilation, implementation, and run-time system,” Jul. 2010

[Malik, 2010] A. Malik, “Principia lingua SystemJ,” The University of Auckland New Zealand, 2010

[Malik et al., 2010] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, “SystemJ: A GALS language for system level design,” *Comput. Lang. Syst. Struct.*, vol. 36, 2010, pp. 317–344

[Mandel & Pouzet, 2005] L. Mandel and M. Pouzet, “ReactiveML: a reactive extension to ML,” *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2005, pp. 82–93

[Maraninchi, 1991] F. Maraninchi, “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems,” *IN IEEE WORKSHOP ON VISUAL LANGUAGES*, 1991

[Massa, 2003] A. J. Massa, *Embedded software development with eCos*, Prentice Hall PTR, 2003

[MathWorks, 2011] MathWorks, "Simulink - Simulation and Model-Based Design," 2011

[Microsoft Corporation, 2008] Microsoft Corporation, "Axum programming language," Sep. 2008

[Milner et al., 1980] R. Milner, R. Milner, R. Milner, and R. Milner, *A calculus of communicating systems*, Springer-Verlag, 1980

[Milner, 1997] R. Milner, *The definition of standard ML: revised*, The MIT press, 1997

[Milner, 1999] R. Milner, *Communicating and mobile systems: the pi-calculus*, Cambridge Univ Pr, 1999

[Le Moigne et al., 2004] R. Le Moigne, O. Pasquier, and J. P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," *Proceedings of the conference on Design, automation and test in Europe-Volume 3*, 2004, p. 30082

[Moores, 1999] J. Moores, "CCSP-A portable CSP-based run-time system supporting C and occam," *Architectures, languages and techniques for concurrent systems: WoTUG-22, proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, 11-14 April 1999, Keele, United Kingdom*, 1999, p. 147

[Nakano et al., 2002] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," *TRON Project International Symposium, 1995., Proceedings of the 12th*, 2002, pp. 34–42

[Nichols et al., 1996] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*, O'Reilly Media, 1996

[Nikaein, 1999] N. Nikaein, *RAMA Reactive Autonomous Mobile Agent*, DEA RSD at ESSI, Sophia Antipolis France, 1999

[Odersky et al., 2004] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, *An overview of the Scala programming language*, 2004

[Oracle, 1999] Oracle, "Java Thread Primitive Deprecation," 1999

[OSC Initiative, 1999] OSC Initiative, *SystemC*, 1999

[OSEK, 1997] OSEK, *Osek/vdx Operating System Specification 2.0*, Jun, 1997

[Passerone et al., 1998] C. Passerone, C. Sansoe, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli, "Modeling reactive systems in Java," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 3, 1998, pp. 515–523

[Peterson, 1977] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, 1977, pp. 223–252

[Petri, 1962] C. A. Petri, "Kommunikation mit automaten," 1962

[Plummer et al., 2006] B. Plummer, M. Khajanchi, and S. A. Edwards, "An Esterel virtual machine for embedded systems," *International Workshop on Synchronous Languages, Applications, and Programming (SLAP)*, 2006

[Poplavko et al., 2003] P. Poplavko, T. Basten, M. Bekooij, J. Van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 63–72

[Posadas et al., 2005] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model," *Design Automation for Embedded Systems*, vol. 10, 2005, pp. 209–227

[POSIX, 2009] POSIX, "Information technology - Portable Operating System Interface (POSIX) Operating System Interface (POSIX)," *ISO/IEC/IEEE 9945 (First edition 2009-09-15)*, 2009

[Potop-Butucaru et al., 2005] D. Potop-Butucaru, R. de Simone, and J. P. Talpin, "The synchronous hypothesis and synchronous languages," *The Embedded Systems Handbook*, 2005

[Potop-Butucaru & Caillaud, 2007] D. Potop-Butucaru and B. Caillaud, "Correct-by-construction asynchronous implementation of modular synchronous specifications," *Fundamenta Informaticae*, vol. 78, 2007, pp. 131–159

[Potop-Butucaru & De Simone, 2003] D. Potop-Butucaru and R. De Simone, "Optimizations for faster execution of Esterel programs," *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, 2003, pp. 227–236

[Pucella, 1998] R. R. Pucella, "Reactive Programming in Standard ML," *Computer Languages, International Conference on*, Los Alamitos, CA, USA: IEEE Computer Society, 1998, p. 48

[Radojevic et al., 2006] I. Radojevic, Z. Salcic, and P. Roop, "Design of heterogeneous embedded systems using DFCharts model of computation," *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, 2006, p. 4

[Radojevic et al., 2006] I. Radojevic, Z. Salcic, and P. S. Roop, "Modeling Embedded Systems: From SystemC and Esterel to DFCharts," *Design & Test of Computers, IEEE*, vol. 23, 2006, pp. 348–358

[Rajan & Shyamasundar, 2000] B. Rajan and R. K. Shyamasundar, "Multiclock ESTEREL: A reactive framework for asynchronous design," *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 201–209

[Ramesh, 1998] S. Ramesh, "Communicating reactive state machines: design, model and implementation," *IFAC Workshop on Distributed Computer Control Systems*, 1998

[Raymond et al., 1998] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber, "Automatic testing of reactive systems," *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, 1998, pp. 200–209

[Ritchie et al., 1975] D. M. Ritchie, B. W. Kernighan, M. Lesk, and inc Bell Telephone Laboratories, *The C programming language*, Bell Laboratories, 1975

[Rochester, 1955] N. Rochester, "The computer and its peripheral equipment," *Papers and discussions presented at the the November 7-9, 1955, eastern joint AIEE-IRE computer conference: Computers in business and industrial systems*, 1955, pp. 64–69

[RTEMS, 2003] C. RTEMS, *Users Guide. Edition 4.6. 5, for RTEMS 4.6. 5*, On-Line Applications Research Corporation (OAR) http://www. rtems. com, 2003

[Salcic et al., 2004] Z. Salcic, P. Roop, M. Biglari-Abhari, and A. Bigdeli, "REFLIX: a processor core with native support for control-dominated embedded applications," *Microprocessors and Microsystems*, vol. 28, 2004, pp. 13–25

[Salcic et al., 2005] Z. Salcic, D. Hui, P. Roop, and M. Biglari-Abhari, "REMIC: design of a reactive embedded microprocessor core," *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 977–981

[Salcic & Mikhael, 2000] Z. Salcic and R. Mikhael, "A new method for instantaneous power system frequency measurement using reference points detection," *Electric Power Systems Research*, vol. 55, 2000, pp. 97–102

[Sangiovanni-Vincentelli & Martin, 2002] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *Design & Test of Computers, IEEE*, vol. 18, 2002, pp. 23–33

[Schneider, 2000] K. Schneider, "A verified hardware synthesis of Esterel programs," *Proceedings of the IFIP WG10*, 2000, pp. 205–214

[Schneider, 2009] K. Schneider, "The synchronous programming language Quartz," *Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Tech. Rep*, 2009

[Séméria & Ghosh, 2000] L. Séméria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++ (short paper)," *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, New York, NY, USA: ACM, 2000, pp. 405–408

[Serrano et al., 2004] M. Serrano, F. Boussinot, and B. Serpette, "Scheme fair threads," *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2004, pp. 203–214

[Shalan & Mooney III, 2002] M. Shalan and V. J. Mooney III, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002, pp. 79–84

[Shiple et al., 1996] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," *Proceedings of the 1996 European conference on Design and Test*, 1996, p. 328

[Silberschatz & Galvin, 1998] A. Silberschatz and P. Galvin, *Operating System Concepts, 5th Edition*, John Wiley & Sons, 1998

[Stepner et al., 1999] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," *Design Automation Conference, 1999. Proceedings. 36th*, 1999, pp. 151–156

[Stroustrup, 2003] B. Stroustrup, *C++*, John Wiley and Sons Ltd., 2003

[Sun & Salcic, 2007] W.-T. Sun and Z. Salcic, "Modeling RTOS for Reactive Embedded Systems," *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems*, IEEE Computer Society, 2007, pp. 534–539

[Susini et al., 1998] J. F. Susini, L. Hazard, and F. Boussinot, "Distributed reactive machines," *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, 1998, pp. 267–274

[Tan et al., 1995] S. Tan, D. K. Raila, W. S. Liao, and R. H. Campbell, *Virtual Hardware for Operating Systems Development*, IEEE TCOS Bulletin, Spring, 1995

[Tomiyama et al., 2001] H. Tomiyama, Y. Cao, and K. Murakami, "Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC," 2001

[Traulsen et al., 2011] C. Traulsen, T. Amende, and R. von Hanxleden, "Compiling SyncCharts to Synchronous C," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, 2011, pp. 1–4

[Wang et al., 2003] G. Wang, P. R. Cook, and others, "ChucK: A concurrent, on-the-fly audio programming language," *Proceedings of International Computer Music Conference*, 2003, pp. 219–226

[Wang et al., 2008] Z. Wang, W. Haberl, S. Kugele, and M. Tautschnig, "Automatic generation of systemc models from component-based designs for early design validation and performance analysis," *Proceedings of the 7th international workshop on Software and performance*, 2008, pp. 139–144

[Welch et al., 2002] P. Welch, J. Aldous, and J. Foster, "CSP networking for java (JCSP. net)," *Computational Science—ICCS 2002*, 2002, pp. 695–708

[Welch & Barnes, 2005] P. H. Welch and F. R. . Barnes, "Communicating mobile processes: introducing occam-pi," *In 25 Years of CSP*, 2005

[Yoo et al., 2002] S. Yoo, G. Nicolescu, L. Gauthier, and A. A. Jerraya, “Automatic generation of fast timed simulation models for operating systems in SoC design,” *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 620–627

[Yoo et al., 2003] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya, “Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer,” *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, 2003, p. 10550

[Yoo & Jerraya, 2005] S. Yoo and A. A. Jerraya, “Hardware/software cosimulation from interface perspective,” *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, 2005, pp. 369–379

[Yoong et al., 2006] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian, “Compiling Esterel for distributed execution,” *International Workshop on Synchronous Languages, Applications, and Programming (SLAP’06), Vienna, Austria*, 2006

[Yuan et al., 2009] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic, “STARPro -- A new multithreaded direct execution platform for Esterel,” *Electronic Notes in Theoretical Computer Science*, vol. 238, Jun. 2009, pp. 37–55

[Zabel et al., 2009] H. Zabel, W. Müller, and A. Gerstlauer, “Accurate RTOS modeling and analysis with SystemC,” *Hardware-dependent Software*, 2009, pp. 233–260