World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:5, No:11, 2011

A Comprehensive and Integrated Framework for Formal Specification of Concurrent Systems

Sara Sharifi Rad, Hassan Haghighi

***Abstract*—**Due to important issues, such as deadlock, starvation, communication, non-deterministic behavior and synchronization, concurrent systems are very complex, sensitive, and error-prone. Thus ensuring reliability and accuracy of these systems is very essential. Therefore, there has been a big interest in the formal specification of concurrent programs in recent years. Nevertheless, some features of concurrent systems, such as dynamic process creation, scheduling and starvation have not been specified formally yet. Also, some other features have been specified partially and/or have been described using a combination of several different formalisms and methods whose integration needs too much effort. In other words, a comprehensive and integrated specification that could cover all aspects of concurrent systems has not been provided yet. Thus, this paper makes two major contributions: firstly, it provides a comprehensive formal framework to specify all well-known features of concurrent systems. Secondly, it provides an integrated specification of these features by using just a single formal notation, i.e., the Z language.

***Keywords*—**Concurrent systems, Formal methods, Formal specification, Z language

I. Introduction

In the recent few years, concurrent processing has been almost everywhere in the computer world. In a concurrent system there exists a set of processes that execute concurrently. Also, each process interacts with other processes based on known approaches. Also, processes interaction is based on competition and/or cooperation. Threads which are in fact lightweight processes present a sample of cooperative processes existing inside a process. Cooperation of threads leads to the increase of concurrency, thereupon multithreading concept is a basic context and extremely useful in concurrent systems [1], [3].

A concurrent system has many possible executions, and its behavior is usually not reproducible [2]. Consequently, the development of concurrent systems is a complex and error-prone task. Therefore, it is useful to specify, develop, and verify concurrent systems using formal methods. To develop a reliable concurrent system, it is significant to deduce relationship between properties of the concurrent system formally because the application of formal methods to the specification of systems is expected to increase the level of confidence in correctness of final programs [5]. In this way, formal methods have been long distinguished about the requirement to formally examine concurrent systems and provide an unambiguous description of these systems [4].

So far several formal specifications of concurrent systems have been presented by various methods and languages (e.g., VCD [14], TLZ [18] and Petri Net [21]). However, many aspects of concurrent systems, such as dynamic process creation, scheduling and starvation, have not been formally specified yet. Also, some other features have been specified partially and/or have been described using a combination of several different formalisms and methods whose integration needs too much effort. In other words, a comprehensive and integrated specification that could cover all aspects of concurrent systems has not been provided yet.

In this paper, we propose a comprehensive framework in order to formally specify all important features of concurrent systems, including *Dynamic process creation*, *Multi-threading*, *Communication*, *Scheduling*, *Mutual exclusion*, *Deadlock,* and *Starvation* using a single notation, i.e., the Z language, which provides us with mathematical techniques needed for specifying, verifying, and refining specifications into code formally. Thus, this paper makes two major contributions: firstly, it provides a comprehensive specification of concurrent systems covering all of their well-known features. Secondly, it provides an integrated specification of these features using a single formal notation, i.e., the Z language.

The paper is organized as follows: in section 2, we review related work. In section 3, a brief survey of *formal methods* and the *Z language* is presented. In section 4, we present our approach to specify concurrent systems. Finally, we conclude the paper in section 5.

II. Related Work

In this section, we point to some related work. As can be seen in Table I, different methods and languages have been so far used to specify various features of concurrent systems. Also, these works do not cover all major aspects of concurrent systems.

Most of existing approaches of concurrent Z specifications have placed emphasis on the use of additional formalisms such as temporal logic, TLA and CSP [9]–[12]. Also, in some papers the behavioral and coordination aspects of concurrent systems are described by combining CCS and Temporal logic and/or GCCS [13], [14]. In this paper, we are going to specify all important aspects of concurrent systems fully based on the Z notation alone.

S. Sh. Faculty of Electrical, Computer and IT Engineering, Islamic Azad University, Qazvin, Iran (phone: 98131-723-1623; fax: 98131-722-6924; e-mail: S.sharifirad@qiau.ac.ir).
H. H. Faculty of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran (e-mail: h_haghighi@sbu. ac.ir).

TABLE I
RELATED WORK TO SPECIFY CONCURRENT SYSTEMS

Feature	**Specification**		
	Aspect specified	***FM***	***Ref. NO***
Communication	Static communications	VCD	[14]
	Process communications	Z	[15]
Scheduling	Real-Time systems scheduling	Z	[16]
Synchronization	Dinning philosophers problem	PZ Z IP STOCS	[17] [18] [19] [20]
Deadlock	Detection / Detection and recovery	PN Z+TL ESL	[21] [18] [22]

III. Overview on the Z Language

Universally, engineers use mathematically based methods to describe systems. Formal method is a technique which employs mathematical notation and possesses a sound mathematical basis. The application of *formal methods* to the specification of software systems is expected to increase the level of confidence in the correctness of final programs [5].

Formal methods need a soundly based specification language. Many languages exist for formal specification; The Z notation, as one of these languages, is an extensive language and has been fostered by its many positive aspects. This specification language is based upon a well-known set theory, namely, Z set theory, and the first order predicate logic. Together, they make up a mathematical language that is easy to learn and to apply [23].

In the Z formal notation, specification constructs (e.g., axiomatic definitions and schemas) are used to modularize the state and behavior of the system being specified. Among these constructs, schema is the most important tool to encapsulate specification chunks. The schema construct is used to model both system state (as state schema) and behavior (as operation schema). A state schema encapsulate (a part of) system state variables with their invariants. An operation schema specifies a possible functionality or behavior on the system state by defining predicates that relate before-state variables (variables before application of the operation) and after-state ones. A valuation of variables in each schema is called its binding set. Most often an Init operation schema is defined on a state schema to define a special binding set as the schema initial state. Then, each operation schema may map a pre-state to an after-state.

The Z language has been so far used to describe the dynamic and non-deterministic behavior of concurrent systems [5], [24]; hence, the capabilities and usefulness of the Z language on concurrent systems have been partially proved; we now show this formalism could operate successfully to model all well-known features of concurrent systems.

IV. Formal Specification of Concurrent Systems Properties

As it has been shown in Table 1, some important aspects of concurrent systems, such as *starvation*, *multi-threading* and *dynamic thread creation* have not been specified yet. In addition, some cases have been specified in a way that is not related to the concurrent system exclusively; for example, the specification of scheduling has been presented for real-time systems not for concurrent systems.

In this section, we propose our comprehensive framework for formal specification of concurrent systems. The first step to achieve the above goal is the presentation of informal specification. More precisely, we provide useful definitions of concurrent system features in part A and then present the related formal specification in part B by referring to associated definitions in part A.

A. Principles of concurrent systems

Presented definitions in this section are derived from the features of concurrent systems [1], [2], [5], [15], [20], [21], [24], [26]–[30]:

Definition 1*: *Concurrency
A concurrent system is a collection of active entities that execute at the same time and interact with each other during their life cycle.

According to Definition 1, concurrent systems are composed of different components called *active entities*. The innuendo of active entity is *process* or *thread*. Each process has a unique name and independent address space. The process life cycle includes *creation*, *scheduling* and *termination*. If at a moment, more than one process is working, then we have indeed concurrency. Processes are execution units which can act in a concurrent manner if they interact with each other in a way that their executions overlap in time and/or there exists a combination of interleaving and overlapping.

A combination of these modes is shown in Fig. 1. In this figure, operations of process I and process II interleave in Time 1, and two operations I″ and II″ overlap in Time 2 because the second operation of process II (i.e., II″) is started before the second operation of process I (i.e., I″) is completed.

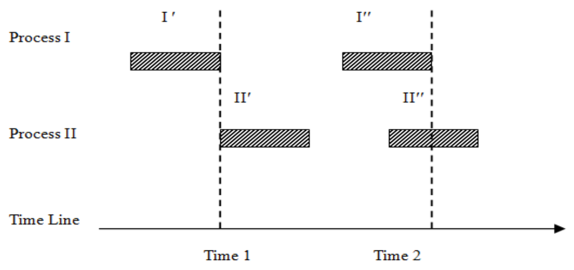

Fig. 1 Combination of interleaving and overlapping

Definition 2: Synchronization
Concurrency introduces the need for communication between executing processes; many resources may be shared between processes and threads in a concurrent system. Then the system requires a means to synchronize their operations.

According to Definition 2, in concurrent systems, both processes and threads need to synchronize among them in order to cooperate effectively when sharing resources or exchanging information. Related to this definition, there is the concept of *critical section*, a code segment in a process that accesses shared resources; these resources may be also accessed by other processes. Only one process must access its critical section at a time [26]. A solution to the critical section problem is *mutual exclusion*.

Definition 3: Coordinator
System resources are maintained and managed by a resource manager, called coordinator.

According to Definition 3, if a process in a concurrent system wants to access a resource, it must send a request message to the coordinator. A key word in concurrent systems is *sharing*: during execution, a resource, such as processor, memory and network, may be shared by various concurrent processes. It means that processes will compete for the resource. Thus, the shared resource must be protected by locking protocols. On the other hand, using the *coordinator* is one of the locking protocols that ensures mutual exclusion (refer to *Definition 2*) for concurrent executions.

Definition 4**: **Dynamic Thread Creation & Multi Threading
Each process, during its execution, can create several threads in own address space.

According to Definition 4, concepts *multithreading* and *dynamic thread creation* are taken. Threads of a process share their parent process address space. Unlike processes, threads do not have their own private address space, but share the state and global variables of a process together with other threads. These aspects have not been yet described formally in the literature.

Definition 5: Non-determinism
A program is non-deterministic if for at least one input, it produces more than one output and/or exhibits more than one behavior [24].

According to Definition 5, concurrent systems inherently exhibit non-deterministic behavior [26]. For example, when several processes compete for the same resource, non-deterministic effects appear [5]. In [24] the notion of multi-schema is defined as a tool for the specifier to specify non-determinism in Z explicitly. In this paper we use the same notation for modeling non-deterministic explicitly.

Definition 6: Communication
Processes need to communicate by passing data between them. Processes can be communicated in two ways: by shared variable *or* message passing.

When processes communicate by *shared variables,* one process "writes" into a variable that is "read" by another process, and when processes communicate by *message passing*, processes are assumed to share a communication *network* and exchange data in messages via "send" and "receive" primitives. Communication by message passing can be either *synchronous* or *asynchronous.*

In *synchronous* communication, communication happens only if the receiving process is waiting for the communication; this is termed a rendezvous. In *asynchronous* point to point message communication, a process sends a message to another process by placing the message in a *location* of network (Unlike the *synchronous* communication which uses networks as communication media, the *asynchronous* communication saves messages into networks); a location is an empty space in the network to hold the message. In an asynchronous communication, it is assumed that each network has an unlimited amount of location so that any number of messages may be placed in the network [15], [26].

Definition 7: Scheduling
During the execution of concurrent systems, fairness must be guaranteed by applying appropriate scheduling.

A scheduling policy is fair if it gives every process that is not delayed chance to proceed. On a single processor system, a scheduling policy is fair if it is unconditionally fair for processes that are not delayed, whereas, on a multi processor system, a scheduling policy is fair if it is unconditionally fair for parallel execution of processes. To specify scheduling in multi processor systems, we use *Gang scheduling* [20, 30] as a typical coscheduling approach that is widely used in concurrent systems. According to this scheduling strategy, a running process does not run forever; it eventually moves to the ready status, giving other processes the chance to proceed.

Definition 8: Standstill
A concurrent system is at the standstill state if no forward progress is being made.

Deadlock and *livelock* situations create standstill conditions for concurrent system [27], [29]. *Deadlock* is the most common problem in concurrent systems, and *livelock* term usually connotes *Starvation* and *Infinite Execution*. The relationship of these concepts is shown in Fig. 2.

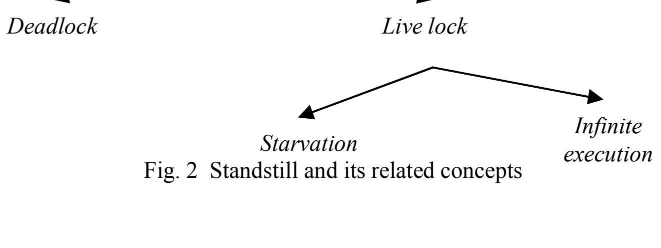

Fig. 2 Standstill and its related concepts

Unlike livelock which may occur for one or more processes, deadlock always occurs for more than one process. Thus, in a deadlock status, whole of system is impaired while in a livelock status, only the current process(es) is (are) impaired. Further explanations are given in the next definitions.

Definition 9: Deadlock
Deadlock is a situation where two or more processes cannot proceed, because they are all waiting for another process to release some resources.

According to Definition 9, deadlock may occur when each process is waiting for another process to perform an operation. In many concurrent systems, the cost of deadlock avoidance is often considerable. Thus, these systems ignore the problems of deadlock until they enter a deadlock state. On the other hand, the occurrence of a deadlock can cripple part of a concurrent system [28]. Consequently, in this article we will consider two approaches to deadlock [27], [28] namely avoidance and detection & recovery approaches. The final decision will be taken in the implementation phase.

Definition 10: Starvation
A process with a non-zero cost may experience starvation.

Starvation may happen when one or more concurrent processes are blocked from gaining access to a resource. Consequently, such processes cannot progress [27], [29]. In this state the process status is *Waiting* continuously.

Definition 11: Infinite Execution
In a concurrent system, a process may execute forever. However, this process cannot progress.

Similar to the starvation status, in this state, the process cannot progress, but unlike the starvation status, in the infinite execution status, the process status exchanges between two statuses *Running* and *Restart* frequently.

B. Formal specification of concurrent systems

In this section, we propose a comprehensive and integrated framework for formal specification of all well-known features of concurrent systems reviewed in the previous subsection including *dynamic thread creation*, *communication*, *scheduling*, *synchronization*, *deadlock, livelock, infinite execution* and *starvation* using the Z language. It is worth mentioning that "Z/eves 2.1" has been used to validate the finally proposed specification. We now present our Z specification of concurrent systems step by step:

[*Address_Space*, *Message*, *PTName*]
The type of address spaces, messages and names of processes are specified by the above *given types* in Z.

According to Definition 6:
Communication_Type:🖬 *MessagePassing* ❁ *SharedVariable*

PT ::🖬 *Process* ❁ *Thread*

Active entities in concurrent systems are processes and threads.
Type_Re ::🖬 *Processor* ❁ *Memory* ❁ *Network*
Type_Re indicates type of resources in the system.

DeadLock_Approach ::🖬 *DetRec* ❁ *AVO*
Answer ::🖬 *Yes* ❁ *No*
According to Definition 9, to obtain a comprehensive specification, we consider both *Deadlock Detection & Recovery* and *Deadlock Avoidance* approaches to deal with deadlock in this paper.
Resource is specified as follows:
✐♉*Resource* ♉♉♉♉♉♉♉♉♉♉♉
✸*type: Type_Re*
✸*Location:* seq *Message*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*type = Processor* ⓿ *Location =* ⇱⇧
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
There is some *Location* in the network and memory as two main resources to hold the messages.
Identifier *type* in *Resource* Schema specifies the type of the resource, and identifier *Location* is specified by a sequence of *Message*.
Type of process or thread operation is specified as follows:
Type_OP ::🖬 *Update* ❁ *ReadOnly* ❁ *Sender* ❁ *Receiver*

Type of process or thread status is specified as follows:
STATUS ::🖬 *Idle*
❁ *Ready*
❁ *Running*
❁ *Finish*
❁ *Restart*
❁ *Waiting*
❁ *Starvation*
❁ *InfiniteExe*

According to Definitions 1 and 4, we use *Pr_Th* schema for Process and Thread specification as follows:
✐♉*Pr_Th*♉♉♉♉♉♉♉♉♉♉♉
✸*Name: PTName*
✸*pt: PT*
✸*NR:* ⚐ *Resource*
✸*EM, IM:* ⚐ *Message*
✸*address: Address_Space*
✸*threadsName:* ⚐ *PTName*
✸*type: Type_OP*
✸*status: STATUS*
✸*PreviousStatuses:* seq *STATUS*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*pt = Thread* ⓿ *threadsName =* 🗀
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
According to Definition 1, each process or thread has a unique Name. Thus, identifier *Name* indicates the unique name of the active entity. Identifier *pt* specifies the type of the active entity (Process or Thread) in the specification. This means that if *pt* is equivalent to Process, then all schema identifiers are related to process features; otherwise, all identifiers are associated to thread features.

NR specifies the set of resources requested by the process or thread right now. *EM* and *IM* show the set of Export and Import messages for each process or thread, respectively. If *pt* is equivalent to *Process*, then *threadsName* shows the set of names of threads which belong to the process. *PreviousStatuses* specifies the sequence of previous statuses of each processes or thread.

According to Definition 3:

✐♉*Coordinator*♉♉♉♉♉♉♉♉♉♉♉♉
✹*Grant: Resource* ▪ *Pr_Th*
✹*queue: Resource* ▪ ⛿ *Pr_Th*
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

The *Coordinator* in our Z specification consists of *Grant* and *queue functions*. According to locking protocols, if a resource is free, the coordinator *Grants* the resource to the requester process; otherwise, the process is added to this resource *queue*.

Now, we specify the state schema of the system as follows:

✐♉*CS* ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹*processes:* ⛿ *Pr_Th*
✹*resources:* ⛿ *Resource*
✹*coordinator: Coordinator*
✹*communication: Pr_Th* ♈ *Pr_Th*
✹*CT: Communication_Type*
✹*DA: DeadLock_Approach*
✹*DL_chance, DL_sure: Answer*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹✌*p: processes* ◎ *p . NR* ⌘ *resources*
✹✌*p, q: processes*
✹ ◎ *q . address = p . address*
✹ ⓿ *p = q* ❖ *p . Name* ♏ *q . threadsName* ❖ *q . Name* ♏ *p . threadsName*
✹✌*p, q: processes* ◎ *p* ⇖ *q* ⓿ *p . Name* ⇖ *q . Name*
✹*CT = SharedVariable*
✹⓿ ☎☜*r: resources* ◎ *r . type = Memory*➀
✹ ❍ ☎✌*p, q: Pr_Th* ✽ ☎*p*✉ *q*➀ ♏ *communication*
✹ ◎ ☎*p . type = Update* ❍ *q . type = ReadOnly*➀➀
✹*CT = MessagePassing*
✹⓿ ☎☜*r: resources* ◎ *r . type = Network*➀
✹ ❍ ☎✌*p, q: Pr_Th* ✽ ☎*p*✉ *q*➀ ♏ *communication*
✹ ◎ ☎*p . type = Sender* ❍ *q . type = Receiver*➀➀
✹✌*r: Resource* ✽ *r* ♏ dom *coordinator . queue*
✹ ◎ ☜*p: Pr_Th* ◎ *p* ♏ *coordinator . queue r* ⓿ *p . status = Waiting*
✹*DA = DetRec* ⓿ *DL_sure* ♏ ❀*Yes*✉ *No*❝
✹*DA = AVO* ⓿ *DL_sure = No*
✹*DL_chance = No* ⓿ *DL_sure = No*
✹*DL_chance = Yes* ⓿ *DL_sure* ♏ ❀*Yes*✉ *No*❝
✹dom *coordinator . Grant* ⌘ *resources*
✹dom *coordinator . queue* ⌘ *resources*
✹ran *coordinator . Grant* ⌘ *processes*
✹✌*p:* ⛿ *Pr_Th* ◎ *p* ♏ ran *coordinator . queue* ⓿ *p* ⌘ *processes*
✹dom *communication* ⌘ *processes*
✹ran *communication* ⌘ *processes*
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

Identifier *Processes* indicates the set of active entities including processes and threads which exist in the concurrent system, and identifier *resources* denotes the set of active resources. *Communication* relationship shows the relevance between each active entity with other active entities. *DL_chance* indicates deadlock possibility among a subset of processes while *DL_sure* determines a deterministic occurrence of deadlock among a subset of processes; we will refer to these identifiers again.

Now we write the initialization schema and all operation schemas of the concurrent system in turn:

✐♉*CSInit* ♉♉♉♉♉♉♉♉♉♉♉
✹*CS'*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹*processes'* = 🗀
✹*resources'* = 🗀
✹*coordinator' . Grant* = 🗀
✹*coordinator' . queue* = 🗀
✹*communication'* = 🗀
✹*DL_chance' = No*
✹*DL_sure' = No*
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

Operation schemas are presented below:

✐♉*Create* ♉♉♉♉♉♉♉♉♉♉♉♉
✹Δ*CS*
✹*p?: Pr_Th*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹*p?* ⬈ *processes*
✹*p? . pt = Process*
✹*p? . NR* ⌘ *resources*
✹*p? . EM* ⌘ *Message*
✹*p? . IM* = 🗀
✹*p? . status = Idle*
✹*p? . PreviousStatuses* = ⇖⇧
✹*processes' = processes* ✞ ❀*p?*❝
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

Create is an operation schema for creating a process in the system. In this schema, the input process (*p?*) will be created and added to the set of system processes.

✐♉*DTC*♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹Δ*CS*
✹*p?: Pr_Th*
✹*new_t?:* ⛿ *PTName*
✹*new_tn!:* ⛿ *PTName*
✹*new_create!:* ⛿ *Pr_Th*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹*p? . status = Running*
✹*p?* ♏ *processes*
✹☜₁*t_set:* ⛿ *Pr_Th*
✹ ◎ *# new_t? = # t_set*
✹ ❍ ☎✌*t: t_set*
✹ ◎ ☎*t . Name* ♏ *new_t?*
✹ ❍ *t . pt = Thread*
✹ ❍ *t . address = p? . address*
✹ ❍ *t . status = Idle*
✹ ❍ *t* ⬈ *processes*➀➀
✹ ⓿ *new_create! = t_set*
✹*new_tn!* = ☎❍*p: processes* ✽ *p = p?* ◎ *p . threadsName* ✞ *new_t?*➀
✹*processes'*
✹ = ❀ *p: processes* ✽ *p* ⇖ *p?* ❝
✹ ✞ ❀ *p: Pr_Th* ✽ *p . threadsName = new_tn!* ❍ *p . Name = p? . Name* ❝
✹ ✞ *new_create!*
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

DTC specifies dynamic thread creation based on Definition 4. Each process can create one or more thread during its running; according to this schema, a set of threads (*new_t?*) will be added to the current threads of the input process (*p?*).

✐♉*Terminate*♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹Δ*CS*
✹*p!: &Pr_Th*
✎♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹☜*p: processes* ◎ *p* ♏ *processes* ❍ *p . status = Finish* ⓿ *p! = p*
✹*coordinator' . Grant = coordinator . Grant* ◆ ❀*p!*❝
✹*processes' = processes* \ ❀*p!*❝
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

Terminate specifies finishing a process in a normal condition. According to definition 5, non-deterministic effects appear in this part of specification since more than one process may have the finish status. Thus, we use the notion of multi-schema (when declaring *p!* by "&") according to the notation given in [24].

✐♉*release*♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹Δ*CS*
✹*p?: Pr_Th*
✹*r!:* ⛿ *Resource*

✸*new_nr!:* 🏱 *Resource*
✸*new_tn!:* 🏱 *PTName*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*p?* ♏ *processes*
✸*p? . status = Restart*
✸*r! = coordinator . Grant*˜ 🡑 ❀*p?*❝ 🡓
✸*coordinator' . Grant = coordinator . Grant* ◆ ❀*p?*❝
✸*new_nr! =* ☎❍*p: processes* ✿ *p = p?* ◎ *p . NR* ✞ *r!*✆
✸*new_tn!*
✸ = ☎❍*p: processes* ✿ *p = p?* ❍ *p . pt = Process*
✸ ◎ *p . threadsName \ p? . threadsName*✆
✸*processes'*
✸ = ❀ *p: processes* ✿ *p* ⬉ *p?* ❝
✸ ✞ ❀ *p: Pr_Th*
✸ ✿ *p . NR = new_nr!*
✸ ❍ *p . threadsName = new_tn!*
✸ ❍ *p . Name = p? . Name* ❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉♉

Release specifies abandonment of all the granted resources to a specific process or thread.

✍♉*ReleaseOneResource* ♉♉♉♉♉♉♉♉♉♉♉♉♉
✸∆*CS*
✸*p?: Pr_Th*
✸*r?: Resource*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*p? . status = Running*
✸☎*r?*✉ *p?*✆ ♏ *coordinator . Grant*
✸*coordinator' . Grant = coordinator . Grant* \ ❀☎*r?*✉ *p?*✆❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

If a process or thread does not need its current resource, then releases it.

✍♉*ND_Req* ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸∆*CS*
✸*r?: Resource*
✸*p!:& Pr_Th*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*p!* ♏ *coordinator . queue r?*
✸*r?* ♏ *resources* \ dom *coordinator . Grant*
✸*coordinator' . Grant = coordinator . Grant* ✞ ❀☎*r?*✉ *p!*✆❝
✸*coordinator' . queue r? = coordinator . queue r?* \ ❀*p!*❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

When several processes compete for the same resource, non-deterministic effects appear since there may exist more than one process which can acquire a specific resource at the same time. Thus, the notion of multi-schema is used for specifying *ND-Req*.

✍♉*Assign_Resource*♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*ND_Req*
✸*new_nr!:* 🏱 *Resource*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*new_nr! =* ☎❍*p: processes* ✿ *p = p!* ◎ *p . NR* \ ❀*r?*❝✆
✸*processes'*
✸ = ❀ *p: processes* ✿ *p* ⬉ *p!* ❝
✸ ✞ ❀ *p: Pr_Th* ✿ *p . NR = new_nr!* ❍ *p . Name = p! . Name* ❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

Assign_Resource includes the schema "*ND-Req*" above to complete the specification of resource allocation to a process existing in the resource queue.

SinScheduling and *CoScheduling* below are scheduling schemas for single-processor systems and multi-processor systems, respectively:

✍♉*SinScheduling*♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*ND_Req*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*r? . type = Processor*
✸*p! . status = Ready*
✸*processes'*
✸ = ❀ *p: processes* ✿ *p* ⬉ *p!* ❝
✸ ✞ ❀ *p: Pr_Th*
✸ ✿ *p . Name = p! . Name*
✸ ❍ *p . pt = p! . pt*
✸ ❍ *p . NR = p! . NR* \ ❀*r?*❝
✸ ❍ *p . EM = p! . EM*
✸ ❍ *p . IM = p! . IM*
✸ ❍ *p . address = p! . address*
✸ ❍ *p . threadsName = p! . threadsName*
✸ ❍ *p . type = p! . type*
✸ ❍ *p . status = Running*
✸ ❍ *p . PreviousStatuses = p! . PreviousStatuses* ♈ ⇱*Ready*⇧ ❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

According to Definition 7, fairness will be guaranteed by a suitable scheduler in the implementation phase, not in the specification stage.

✍♉*CoScheduling* ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸∆*CS*
✸*p?: Pr_Th*
✸*r_set?:* 🏱 *Resource*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸*p?* ♏ *processes*
✸*p? . status = Ready*
✸✌*r: resources* ✿ *r* ♏ *r_set?* ◎ *r . type = Processor*
✸*r_set?* ⌘ *resources* \ dom *coordinator . Grant*
✸# *p? . threadsName* = # *r_set?*
✸✌*r: Resource* ✿ *r* ♏ *r_set?*
✸ ◎ *coordinator' . Grant = coordinator . Grant* ✞ ❀☎*r*✉ *p?*✆❝
✸*processes'*
✸ = ❀ *p: processes* ✿ *p* ⬉ *p?* ❝
✸ ✞ ❀ *p: Pr_Th*
✸ ✿ *p . Name = p? . Name*
✸ ❍ *p . pt = p? . pt*
✸ ❍ *p . NR = p? . NR \ r_set?*
✸ ❍ *p . EM = p? . EM*
✸ ❍ *p . IM = p? . IM*
✸ ❍ *p . address = p? . address*
✸ ❍ *p . threadsName = p? . threadsName*
✸ ❍ *p . type = p? . type*
✸ ❍ *p . status = Running*
✸ ❍ *p . PreviousStatuses = p? . PreviousStatuses* ♈ ⇱*Ready*⇧ ❝
✌♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉

According to Definition 7, in *CoScheduling* schema, dependent processes are gangs scheduled to run simultaneously on distinct processors. Gangs are scheduled to run at the same time. Each process consists of a number of interacting threads.

✍♉*SLS*♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸∆*CS*
✸*r?: Resource*
✸*hun_p!: Pr_Th*
⮰♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✸☜*p: processes* ✿ *p* ♏ *coordinator . queue r?*
✸ ◎ *p . PreviousStatuses* ⬉ ⇱⇧
✸ ❍ # *p . PreviousStatuses* ✇ 1
✸ ❍ ☎✌*i:* 1 .. # *p . PreviousStatuses* ◎ *p . PreviousStatuses i = Waiting*✆
✸ ❶ *hun_p! = p*
✸*processes'*
✸ = ❀ *p: processes* ✿ *p* ⬉ *hun_p!* ❝
✸ ✞ ❀ *p: Pr_Th*
✸ ✿ *p . Name = hun_p! . Name*
✸ ❍ *p . pt = hun_p! . pt*
✸ ❍ *p . NR = hun_p! . NR*
✸ ❍ *p . EM = hun_p! . EM*
✸ ❍ *p . IM = hun_p! . IM*
✸ ❍ *p . address = hun_p! . address*
✸ ❍ *p . threadsName = hun_p! . threadsName*

```
✹        ❍ p . type = hun_p! . type
✹        ❍ p . status = Starvation
✹        ❍ p . PreviousStatuses = hun_p! . PreviousStatuses ♈
⇱Waiting⇧  “
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

According to Definition 8 and Fig.2, *SLS* schema specifies Standstill-Livelock-Starvation state. According to Definition 10, if all previous statuses of a process are *Waiting*, then the process status is starvation.

```
✐♉SLI♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹ΔCS
✹p?: Pr_Th
✹shift_amount?, length!: ☠
⇦♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹p? ♏ processes
✹p? . status = Restart
✹p? . PreviousStatuses ⇱ ⇱⇧
✹length! = # p? . PreviousStatuses
✹1 ⬂ shift_amount? ⬂ length! - 1
✹length! - shift_amount? + 1 mod 2 = 0
✹p? . PreviousStatuses shift_amount? = Running
✹❍ p? . PreviousStatuses length! = Restart
✹✌i: 1 .. length!
✹  ◎  2 * i - 2 + shift_amount? ⬂ length! - 1
✹    ❍ p? . PreviousStatuses ☎2 * i - 2 + shift_amount?✆ = Running
✹    ❍ 2 * i - 1 + shift_amount? ⬂ length!
✹    ❍ p? . PreviousStatuses ☎2 * i - 1 + shift_amount?✆ = Restart
✹processes'
✹   = ❀   p: processes ✿   p ⇱ p?   “
✹    ✞ ❀   p: Pr_Th
✹        ✿   p . Name = p? . Name
✹        ❍ p . pt = p? . pt
✹        ❍ p . NR = p? . NR
✹        ❍ p . EM = p? . EM
✹        ❍ p . IM = p? . IM
✹        ❍ p . address = p? . address
✹        ❍ p . threadsName = p? . threadsName
✹        ❍ p . type = p? . type
✹        ❍ p . status = InfiniteExe
✹        ❍ p . PreviousStatuses = p? . PreviousStatuses ♈ ⇱Restart⇧   “
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

According to Definition 8 and Fig.2, *SLI* schema specifies Standstill-Livelock-Infinite execution. Now according to Definition 8, livelock situation is specified as follows:

```
✐♉LiveLock ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹SLS
✹SLI
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

```
✐♉CircularCondition♉♉♉♉♉♉♉♉♉♉♉
✹ΔCS
✹p?: Pr_Th
✹r?: Resource
✹len_set!: ☠
⇦♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹r? ♏ resources
✹☜p_set: seq processes
✹  ◎   len_set! = # p_set
✹    ❍ p? = p_set len_set!
✹    ❍ ☎✌i: 1 .. len_set! - 1
✹        ◎   ☎☜r: resources ✿   r ⇱ r?
✹            ◎   ☎☎r📪 p_set ☎i + 1✆✆ ♏ coordinator . Grant
✹                ❍ ☎r📪 ❀p_set i“✆ ♏ coordinator . queue✆✆✆
✹    ❍ ☎r?📪 p_set 1✆ ♏ coordinator . Grant
✹    ❍ r? ♏ p? . NR
✹DL_chance' = Yes
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

CircularCondition checks deadlock possibility in a subset of processes. The output of this schema is either Yes or No.

```
✐♉Synchronization♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉
✹CircularCondition
⇦♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹p? ♏ processes
✹p? . status ↗ ❀Finish📪 Waiting📪 Restart“
✹r? ♏ p? . NR
✹r? ♏ resources \ dom coordinator . Grant
✹ʊ coordinator' . Grant = coordinator . Grant ✞ ❀☎r?📪 p?✆“
✹processes'
✹   = ❀   p: processes ✿   p ⇱ p?   “
✹    ✞ ❀   p: Pr_Th
✹        ✿   p . Name = p? . Name
✹        ❍ p . pt = p? . pt
✹        ❍ p . NR = p? . NR \ ❀r?“
✹        ❍ p . EM = p? . EM
✹        ❍ p . IM = p? . IM
✹        ❍ p . address = p? . address
✹        ❍ p . threadsName = p? . threadsName
✹        ❍ p . type = p? . type
✹        ❍ p . status ♏ ❀Ready📪 Running“
✹        ❍ p . PreviousStatuses = p? . PreviousStatuses ♈ ⇱p? . status⇧   “
✹r? ♏ dom coordinator . Grant
✹ʊ DA = DetRec ❍ DL_chance ♏ ❀Yes📪 No“ ❖ DA = AVO ❍ DL_chance = No
✹  ʊ coordinator' . queue r? = coordinator . queue r? ✞ ❀p?“
✹processes'
✹   = ❀   p: processes ✿   p ⇱ p?   “
✹    ✞ ❀   p: Pr_Th
✹        ✿   p . Name = p? . Name
✹        ❍ p . pt = p? . pt
✹        ❍ p . NR = p? . NR
✹        ❍ p . EM = p? . EM
✹        ❍ p . IM = p? . IM
✹        ❍ p . address = p? . address
✹        ❍ p . threadsName = p? . threadsName
✹        ❍ p . type = p? . type
✹        ❍ p . status = Waiting
✹        ❍ p . PreviousStatuses = p? . PreviousStatuses ♈ ⇱p? . status⇧   “
✹DA = DetRec ❍ DL_chance = Yes ʊ DL_sure' = Yes
✹DA = AVO ❍ DL_chance = Yes ʊ coordinator' . queue r? = coordinator . queue r?
✹processes'
✹   = ❀   p: processes ✿   p ⇱ p?   “
✹    ✞ ❀   p: Pr_Th
✹        ✿   p . Name = p? . Name
✹        ❍ p . pt = p? . pt
✹        ❍ p . NR = p? . NR
✹        ❍ p . EM = p? . EM
✹        ❍ p . IM = p? . IM
✹        ❍ p . address = p? . address
✹        ❍ p . threadsName = p? . threadsName
✹        ❍ p . type = p? . type
✹        ❍ p . status = Restart
✹        ❍ p . PreviousStatuses = p? . PreviousStatuses ♈ ⇱p? . status⇧   “
✍♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉
```

According to Definition 2, processes and threads need to be synchronized. In *Synchronization* schema, synchronization is done based on two types of deadlock approaches.

```
✐♉DeadLock_Recovery♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉♉
✹Synchronization
✹p_loop?: ⚐ Pr_Th
✹p!:& Pr_Th
⇦♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹DA = DetRec
✹DL_sure = Yes
✹✌p: Pr_Th ✿   p ♏ p_loop? ◎   p . status = Waiting
✹p! ♏ p_loop?
✹processes'
✹   = ❀   p: processes ✿   p ⇱ p!   “
✹    ✞ ❀   p: Pr_Th
✹        ✿   p . Name = p! . Name
✹        ❍ p . pt = p! . pt
✹        ❍ p . NR = p! . NR
✹        ❍ p . EM = p! . EM
✹        ❍ p . IM = p! . IM
✹        ❍ p . address = p! . address
✹        ❍ p . threadsName = p! . threadsName
✹        ❍ p . type = p! . type
✹        ❍ p . status = Restart
✹        ❍ p . PreviousStatuses = p? . PreviousStatuses ♈ ⇱Waiting⇧   “
✹DL_sure' = No
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

If deadlock approach is detection & recovery, then it is resolved by killing a process or thread existing in the detected cycle randomly; hence, we used the notion of multi-schema when specifying *DeadLock_Recovery*.

```
✍♉Asynchronous_Communication ♉♉♉♉♉♉♉♉♉♉
✹ΔCS
✹p?: Pr_Th
✹r?: Resource
✹new_M!: ⚐ Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹CT = MessagePassing
✹r? . type = Network
✹☎r?🖃 p?☽ ♏ coordinator . Grant
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉
```

According to Definition 6, in the asynchronous message passing, a message can be placed on a location of the network, provided there is some empty space in the network to hold the message; it is assumed that each network has an unlimited amount of space. Operation schemas *As_Send_Me* and *As_Receive_Me* below specify sending and receiving messages operations, respectively.

```
✍♉As_Send_Me ♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹Asynchronous_Communication
✹m?: Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹☞q: processes ◎    ☎p?🖃 q☽ ♏ communication
✹m? ♏ p? . EM
✹☎r?🖃 p?☽ ♏ coordinator . Grant
✹new_M! = ☎❍p: processes ❁    p = p? ◎    p . EM \ ❀m?“☽
✹processes'
✹  = ❀    p: processes ❁    p ⇱ p?   “
✹    ✞ ❀    p: Pr_Th ❁    p . EM = new_M! ❍ p . Name = p? . Name   “
✹resources'
✹  = ❀    r: resources ❁    r ⇱ r?   “
✹    ✞ ❀    r: Resource
✹         ❁    r . type = r? . type ❍ r . Location = r? . Location ♈ ⇱m?⇧   “
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉♉♉♉
✍♉As_Receive_Me♉♉♉♉♉♉♉♉♉♉♉♉♉
✹Asynchronous_Communication
✹m!: Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹☞q: processes ◎    ☎q🖃 p?☽ ♏ communication
✹r? . Location ⇱ ⇱⇧
✹m! = head r? . Location
✹new_M! = ☎❍p: processes ❁    p = p? ◎    p . IM ✞ ❀m!“☽
✹processes'
✹  = ❀    p: processes ❁    p ⇱ p?   “
✹    ✞ ❀    p: Pr_Th ❁    p . IM = new_M! ❍ p . Name = p? . Name   “
✹resources'
✹  = ❀    r: resources ❁    r ⇱ r?   “
✹    ✞ ❀    r: Resource
✹         ❁    r . type = r? . type ❍ r . Location = tail r? . Location   “
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✍♉Synchronous_SeAndRe♉♉♉♉♉♉♉♉♉♉
✹ΔCS
✹p?, q?: Pr_Th
✹m?: Message
✹new_pm!, new_qm!: ⚐ Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹CT = MessagePassing
✹☎p?🖃 q?☽ ♏ communication
✹m? ♏ p? . EM
✹new_pm! = ☎❍p: processes ❁    p = p? ◎    p . EM \ ❀m?“☽
✹new_qm! = ☎❍q: processes ❁    q = q? ◎    q . IM ✞ ❀m?“☽
✹processes'
✹  = ❀    p: processes ❁    p ⇱ p? ❍ p ⇱ q?   “
✹    ✞ ❀    p: Pr_Th ❁    p . EM = new_pm! ❍ p . Name = p? . Name   “
✹    ✞ ❀    q: processes ❁    q . IM = new_qm! ❍ q . Name = q? . Name   “
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉♉♉♉
```

Synchronous_SeAndRe schema specifies synchronous message passing. According to Definition 6, in the synchronous message passing, the sender process delays until the receiving process is ready to receive the message. Messages do not have to be saved in a location of the network.

Communication via shared variables is specified as follows:

```
✍♉SharedMemory_Communication ♉♉
✹ΔCS
✹p?: Pr_Th
✹r?: Resource
✹new_M!: ⚐ Message
✹new_r!: Resource
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹CT = SharedVariable
✹r? . type = Memory
✹☎r?🖃 p?☽ ♏ coordinator . Grant
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✍♉Write_Message♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉
✹SharedMemory_Communication
✹m?: Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹☞q: processes ◎    ☎p?🖃 q☽ ♏ communication
✹m? ♏ p? . EM
✹new_M! = ☎❍p: processes ❁    p = p? ◎    p . EM \ ❀m?“☽
✹processes'
✹  = ❀    p: processes ❁    p ⇱ p?   “
✹    ✞ ❀    p: Pr_Th ❁    p . EM = new_M! ❍ p . Name = p? . Name   “
✹resources'
✹  = ❀    r: resources ❁    r ⇱ r?   “
✹    ✞ ❀    r: Resource
✹         ❁    r . type = r? . type ❍ r . Location = r? . Location ♈ ⇱m?⇧   “
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
♉♉♉♉
✍♉Read_Message ♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹SharedMemory_Communication
✹m!: Message
☜♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
✹☞q: processes ◎    ☎q🖃 p?☽ ♏ communication
✹r? . Location ⇱ ⇱⇧
✹m! = head r? . Location
✹new_M! = ☎❍p: processes ❁    p = p? ◎    p . IM ✞ ❀m!“☽
✹processes'
✹  = ❀    p: processes ❁    p ⇱ p?   “
✹    ✞ ❀    p: Pr_Th ❁    p . IM = new_M! ❍ p . Name = p? . Name   “
✹resources'
✹  = ❀    r: resources ❁    r ⇱ r?   “
✹    ✞ ❀    r: Resource
✹         ❁    r . type = r? . type ❍ r . Location = tail r? . Location   “
ℨ♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉♉
```

According to Definition 6, in shared memory systems, processes/threads communicate together using two operations write and read on shared variables; these operations are similar to send and receive in the asynchronous communication.

V. Conclusion and Future Work

In this paper, at first the well-known aspects of concurrent systems including *dynamic thread creation*, *communication*, *scheduling*, *synchronization*, *deadlock, livelock, infinite execution* and *starvation* were reviewed informally. Then, a comprehensive and integrated framework for formal specification of these aspects was provided using the Z formal specification language. The final specification has been validated using a well-known Z type checker, i.e., Z/eves 2.1. In summary, we can assert that this paper benefits from the following advances in comparison to related work in the literature:

1. This work covers all well-known aspects of concurrent systems whereas some features of these systems, such as dynamic process creation, scheduling, and starvation, have not been specified formally yet.
2. Some other features of concurrent systems have been so far specified partially and/or have been described using a combination of several different formalisms and methods whose integration needs too much effort; see a detailed description in section 2; however, the formal specification proposed in this paper is fully based on a single formal specification language, i.e., the Z notation.
3. Unlike many other approaches in the literature, the work of this paper brings non-determinism into the specification explicitly. As it can be found in the previous section, we used the notion of multi-schema whenever we had to specify non-deterministic behavior. According to the discussion given in [5], such a specification leads to a program which preserves all allowable behaviors of the specified concurrent system.

One of the most important aims of specifying applications formally is to develop programs from formal specifications. We have chosen Z since it has an interpretation in Martin-Löf's theory of types [31]. Therefore, as a future work, we are going to use this interpretation in order to translate our Z specification of a concurrent program into its counterpart in Martin-Löf's theory of types and then drive a functional program from a correctness proof of the resulting type theoretical specification. In this way, we can provide a completely formal way to specify and develop concurrent systems.

References

[1] P. Brinch Hansen, "Operating System Principles," Prentic-Hall,1973.

[2] J. Bacon, J. Van der Linden, "Concurrent Systems: an integrated approach to operating systems, distributed systems and databases," 3nd Edition, *international computer science series*, 2002.

[3] A.J. Bijoy, D.P. Hiren, "Generating Multi-Threaded Code from Polychronous Specifications," *ElsevierJournal, Electronic Notes In Theoretical Computer Science*, vol. 238, 2009, pp. 57-69.

[4] S.C. Harpreet, W.B John, and M.W Jeanette, " Formal Specification of Concurrent Systems," *Elsevier Journal, Advances In Engineering Software*, vol. 30, 1999, pp. 211-224.

[5] H. Haghighi, "Towards a Formal Framework for Developing Concurrent Programs: Modeling Dynamic Behavior," *Proc. The eighth ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-10)*, Hammamet, Tunisia, 2010.

[6] O. Mosbahi, L. Jemni Ben Ayed, and M. Khalgui ,"A Formal Approach for The Development of Reactive Systems," *Elsevier Journal, Information and Software Technology*,vol. 53, pp. 14-33, 2011.

[7] N. Aoumeur, K. Barkaoui, and G. Saake, "Towards MAUDE-TLA based Foundation for Complex Concurrent Systems Specification and Certification," *IEEE Fifth International Conference on Information Technology: New Generation*, 2008.

[8] M. Yusufa, G. Yusufu, "Comparison of SoftwareSpecification Methods Using a Case Study," *IEEE International conference on computer science and software Engineering*, 2008.

[9] R. Duke, I. J. Hayes, P. King, and G. A. Rose, "Protocol Specification and Verification Using Z" *In* IFIP Eighth International Workshop on Protocol Specification, Testing and Verification, *North-Holland*, 1988, pp. 33-46.

[10] E. Fergus, D. Ince, "Z Specifications and Modal Logic," *Proceedings of Software Engineering 90, Brighton*, Ed. Patrick Hall, Cambridge University Press, July 1990.

[11] L. Lamport, "TLZ," *Proceeding of the 8th Z Users Meeting*, Cambridge, Springer Verlage, 1994.

[12] J.C.P Woodcook, and C. Morgan, "Refinement of State-Based Concurrent Systems," *Procs. Of VDM 90*, Springer Verlag, 1990,pp.341-351

[13] D. Safranek, "Visual Specification of Concurrent Systems," *IEEE International Conference on Automated Software Engineering*, 2003.

[14] D. Safranek, "Visual Specification of Systems with Heterogeneous Coordination Models," *Elsevier Electronic Notes in theoretical computer Science*, 2007, pp. 107-121.

[15] A.S. Evans, "Specifying & Verifying Concurrent Systems Using Z," *In: ISCIS XI*, Turkey 1994.

[16] M. Pilling, A. Buruns, and K. Raymond, "Formal Specification and Proof of Inheritance Protocols for Real_Time Scheduling," *IEEE Software Engineering Journal*, vol. 5, September 1990, pp.236-279.

[17] X. He, "PZ nets_a formal method integrating petrinets whit Z," *Elsevier Information and Software Technology*, vol.43 ,2001, pp.1-18.

[18] P. Stocks, K. Raymond, D. Carrington, and A. Lister, "Modelling Open Distributed Systems in Z," *Elsevier computer Communications*, vol.15, March1992, pp. 103-113.

[19] C. Chu Chiang, "Development of Concurrent Systems Through Coordination," *IEEE International Conference on Information Technology*, 2005.

[20] V. Kumar Garg, "Specification and Analysis of Concurrent Systems Using STOCS model," *IEEE Computer Networking Symposium*, 1988.

[21] D.E. Cook, "Formal Specification of Resource-Deadlock Prone Petri Net," *Elsevier Systems Software Journal*, vol.11, 1990, pp.53-69.

[22] N.D. Francesco, G. Vaglini, "Modular Verification of Correctness Properties in Enviorment for Concurrent Systems Specification Deadlock Case," *Elsevier Information Software Technology*, vol.32, October 1990, pp.133-148.

[23] J. Woodcock, J. Davies, "Using Z, Specification, Refinment and Proof," Prentic Hall, 1996.

[24] H. Haghighi, S.H, Mirian-Hosseinabadi, "Nondeterminism in Constructive Z," *Fundamenta Informatica*, Vol.88, 2008, pp. 109-134.

[25] V. Varadharjan, "Use of a Formal Description Technique in the Specification of Authentication Protocols," *Elsevier Computer Standards and Interfaces*,vol. 9, 1990, pp.203-215.

[26] E. Spiliopoulou, "Concurrent and Distributed Functional Systems," PhD Thesis, Department of Computer Science, University of Bristol, 1999.

[27] H. Alex, S.Steven , and H. Steven, "On Deadlock,Livelock and Forward Progress," Technical Reports, university of cambridge, 2005.

[28] M.N. YousufAli, and M.Z.H. Sarker, "An Algorithm for Avoiding Deadlock,"*IEEE INMIC, 9th International Multitopic Conference,* 2005.

[29] K.Ch. Tai, "Definition and Detection of Deadlock, LiveLock, and Starvation in Concurrent Programs," *IEEE Computer Society, International Conference on Parallel Processing*, vol.2, 1994, pp.69-72.

[30] H.D. Karatza, "Scheduling Gang in a Distributed System," *ninth IEEE Workshop ,I.J. of simulation*, May 2003, pp.15-22.

[31] S.H. Mirian-Hosseinabadi, "Constructive Z," Ph.D. dissertation, Essex Univ., 1997.