A Formal Framework for Specifying Concurrent Systems

Sara Sharifirad
Department of Electrical, Computer and IT Engineering, Qazvin Branch
Islamic Azad University
Qazvin, Iran

Hassan Haghighi
Faculty of Electrical and Computer Engineering
Shahid Beheshti University G.C.
Tehran, Iran

ABSTRACT
Concurrent systems are very complex and error-prone because these systems are associated with significant issues, such as deadlock, starvation, communication, non-deterministic behavior and synchronization. Using formal methods, which are based on mathematical notions and theories, can help to increase confidence in these systems. Thus in recent years, most efforts have focused to specify, verify and develop concurrent systems formally. However, with specifications that have been done up to this time, several important aspects of concurrent systems, such as dynamic process creation, scheduling, starvation and infinite execution have not been specified formally yet. On the other hand, some specified aspects, such as deadlock, synchronization and communication have not been described as completely and accurately and/or have been specified using a combination of several different methods and formalisms so that the integration of existing specifications needs too much effort. It can be said unequivocally that until now there is no specification framework, based on a single formalism, for concurrent systems in which all important aspects of these systems are considered. Thus, our previous work tried to present an integrated formal specification framework of all the extracted aspects based on just one formal notation, i.e., the Z language. In this paper, the details of the mentioned formal framework are first presented. Then, this framework is evaluated from two viewpoints: comprehensiveness of the framework itself and appropriateness of Z to write an integrated formal specification of concurrent systems.

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

1. INTRODUCTION
The type of process interaction in concurrent systems has led to specific features of these systems, e.g., deadlock, starvation, scheduling and synchronization. On the other hand, threads which are in fact lightweight processes present a sample of cooperative processes existing inside a process. Cooperation of threads increases concurrency, thereupon multithreading concept is a basic context in concurrent systems [1], [2].

A concurrent system has many possible executions, and its behavior is usually not reproducible [3]. Unfortunately, this feature will cause the development of concurrent systems to be done in the case of risk. 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 the correctness of final programs [4]. In this way, formal methods have been long distinguished for the requirement to formally examine concurrent systems and provide an unambiguous description of these systems [5].

Although different types of formal languages/methods, such as VCD [6], TLZ [7] and Petri Net [8], have been used to specify concurrent systems, many aspects of them, such as dynamic process/thread creation, scheduling, infinite execution and starvation have not been specified formally yet. In addition, other important aspects of concurrent systems have been specified partially and/or have been specified using a combination of several different methods and formalisms whose integration needs too much effort. These limitations not only deprive us of having a comprehensive, formal specification of concurrent systems, but also prevent us from verifying and developing these systems formally in a simple and cost-effective manner. Therefore, in our previous work [9], it was tried to provide a formal specification framework, which covers all the important aspects of concurrent systems.

Also, the proposed framework was only based on a unique formalism. Because of the following reasons, the Z notation was used to specify concurrent systems:

1. Z has a long history in academic and industrial areas.
2. Z is based on set theory and first order predicate logic both of which are easy to be learnt and applied [10].
3. There are several well-known methods to verify Z specifications and develop programs from these specifications.
4. A main part of Z pertains to the schema notation which facilitates the specification of large systems.
5. The notion of nondeterminism, which exists in the behavior of concurrent systems inherently, has been already modeled in Z [11].

This paper aims at:

1. Presenting the details of the mentioned framework.
2. Evaluating the framework based on two goals: comprehensiveness of the framework itself and appropriateness of the used formalism, i.e., Z, to write an integrated formal specification of concurrent systems.

The paper is organized as follows: section 2 reviews related work. Section 3 presents the approach of this work to specify concurrent systems. Evaluation of the given formal framework is presented in section 4. Finally, the paper is concluded in section 5.

2. RELATED WORK
As can be seen in Table 1, *different* methods and languages have been so far used to specify various aspects of concurrent systems. Also, these approaches 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 [12]–[14]. Also, in some papers the behavioral and coordination aspects of concurrent systems are described by combining CCS and Temporal logic and/or GCCS [15], [6]. In this paper, all important aspects of concurrent systems are going to be specified fully based on the Z notation alone.

Table 1. Related work to specify concurrent systems

Issue	*Specified Aspect*	*Formalism*	*Ref. NO*
Communication	Static communications	VCD	[6]
	Process communications	Z	[16]
	Buffer case study	Circus	[17]
Scheduling	Real-Time systems scheduling	Z	[18]
Synchronization	Dinning philosophers problem	PZ	[19]
		Z	[7]
		IP	[20]
		STOCS	[21]
	Buffer case study	Circus	[17]
Deadlock	Detection/ Detection and Recovery	PN	[8]
		Z+TL	[7]
		ESL	[22]

3. SPECIFICATION FRAMEWORK FOR CONCURRENT SYSTEMS

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 formally 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, the details of the comprehensive and integrated framework given in [9] for formal specification of concurrent systems are presented. The formal specification is presented by referring to associated definitions in [9]. It is worth mentioning that "Z/eves 2.1" has been used for type checking of the finally proposed specification. The Z specification of concurrent systems is now presented step by step:

[*Address_Space*, *Message*, *PTName*]

The type of address spaces, messages and names of processes (and threads) are specified by the above *given types* in Z. *Address_Space*, *Message and PTName* represent the maximal set of all address spaces, messages and unique name of the active entities, respectively.

According to the definition of "communication" in [9]:

Communication_Type:= *MessagePassing* | *SharedVariable*

The type of system communication is specified by *Communication_Type.*

According to the definition of "concurrency" in [9]:

AE ::= *Process* | *Thread*

Active Entities (*AE*) in concurrent systems are processes and threads. Since most properties of processes and threads are the same, so they can be named as active entities. *AE* specifies the type of an active entity in the system.

Type_Re ::= *Processor* | *Memory* | *Network*

Type_Re specifies the type of a resource in the system. The most common resources used in the system, i.e., *processor*, *memory* and *network* are intended.

DeadLock_Approach ::= *DetRec* | *AVO*

IsLoop ::= *Yes* | *No*

According to the definition of "deadlock" in [9], to obtain a comprehensive specification, both *Deadlock Detection & Recovery* and *Deadlock Avoidance* approaches to deal with deadlock are considered in this paper. Also by *IsLoop*, presence or absence of loop in subset of active entities will be determined.

Resource is specified as follows:

Resource

Type: Type_Re; Location: seq *Message*

Type = Processor ⇒ *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 associated with a sequence of *Messages*. By the constraint section of *Resource* schema, it is emphasized that *Processors* do not hold any messages.

Type of process or thread operation is specified as follows:

Type_OP ::= *Update* | *ReadOnly* | *Sender* | *Receiver*

Generally, the operation of the process or thread is divided into four types: *Update* and *ReadOnly* are related to the operations of writing type and reading type on shared memory, respectively. *Sender* and *Receiver* are respectively related to send and receive operations in message passing systems.

Type of process or thread status is specified as follows:

STATUS ::= *Idle*|*Ready*|*Running*|*Finish*|*Restart*
|*Waiting*|*Starvation*|*InfiniteExe*

The application of other states, i.e., *Restart*, *Starvation* and *InfiniteExe* will be determined during the system operations.

According to definitions of "concurrency", "dynamic thread creation" and "multi-threading" in [9], Pr_Th schema is used for Process and Thread specification as follows:

Pr_Th

Name: PTName; PT: AE; NR: ℙ *Resource*
EM, IM: ℙ *Message; Address: Address_Space*
ThreadsName: ℙ *PTName; Type: Type_OP*
Status: STATUS; PreviousStatuses: seq *STATUS*

PT = Thread ⇒ *ThreadsName* = ∅

Each process or thread has a unique Name. Thus, *Name* indicates the unique name of the active entity. *PT* specifies the type of the active entity (Process or Thread). 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 *Process*, then *ThreadsName* shows the set of names of threads which belong to the process. *PreviousStatuses* specifies the sequence of previous statuses of each process or thread.
According to the definition of "coordinator" in [9]:

Coordinator

Grant: Resource ⇸ *Pr_Th; queue: Resource* ⇸ ℙ *Pr_Th*

Coordinator plays an important role in the formal specification given in this work: Active entities should be synchronized to use shared resources. Often the synchronization action of competing active entities is done by locking protocols. Here, *Coordinator* is used to support locking protocols. *Coordinator* 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*.
To specify the range of *Queue* function, *power set* (i.e., ℙ) is used instead of the *sequence* (i.e., seq) because priority should not be considered in the queue of resources in the specification time. In other words, priority is an implementation issue.
Now, the state schema of the system is specified as follows:

CS

processes: ℙ *Pr_Th; resources:* ℙ *Resource*
coordinator: Coordinator; communication: Pr_Th ↔ *Pr_Th*
CT: Communication_Type; DA: DeadLock_Approach
DL_chance, DL_sure: IsLoop

∀*p: processes* • *p. NR* ⊆ *resources*
∀*p, q: processes* • *q. Address* = *p. Address*
⇒ *(p* = *q* ∨ *p. Name* ∈ *q. ThreadsName* ∨ *q. Name* ∈ *p. ThreadsName)*
∀*p, q: processes* • *p* ≠ *q* ⇒ *p. Name* ≠ *q. Name*
CT = SharedVariable ⇒ (∃*r: resources* • *r .Type = Memory*)
(∀*p, q: Pr_Th* | (*p, q*)∈ *communication*
• (*p. Type = Update* ∧ *q. Type = ReadOnly*))
CT = MessagePassing ⇒ (∃*r: resources* • *r. Type = Network*)
(∀*p, q: Pr_Th* | (*p, q*)∈ *communication*
• (*p. Type = Sender* ∧ *q. Type = Receiver*))
∀*r: Resource* | *r* ∈dom *coordinator. Queue*
• ∃*p: Pr_Th* • *p* ∈ *coordinator. Queue r* ⇒ *p. Status = Waiting*
DA = DetRec ⇒ *DL_sure* ∈{*Yes, No*}
DA = AVO ⇒ *DL_sure = No* ∧
DL_chance = No ⇒ *DL_sure = No*
DL_chance = Yes ⇒ *DL_sure* ∈{*Yes, No*}
dom *coordinator. Grant* ⊆ *resources*
dom *coordinator. Queue* ⊆ *resources*
ran *coordinator. Grant* ⊆ *processes*
∀*p:* ℙ *Pr_Th* • *p* ∈ran *coordinator. Queue* ⇒ *p* ⊆ *processes*
dom *communication* ⊆ *processes* ∧
ran *communication* ⊆ *processes*

The most important identifiers used in the state schema are as follows: *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.
In the constraint part of the state schema, all theoretical assumptions are expressed as formal. For example, processes or threads should be known by each other to communicate. Now, if the type of communication is selected as *SharedVariable* in the system from the beginning, then process or thread operation *Update* and *ReadOnly* is considered. However, if the type of communication is selected as *MessagePassing* from the beginning, then process or thread operation *Sender* and *receiver* is considered. On the other hand, if the *Detection & Recovery* approach is selected for deadlock problem in the system, then *Dl_sure* value can be *Yes* or *No*; otherwise (i.e., when *Avoidance* approach is used), *Dl_sure* value is *No*.
Here is the initialization schema:

CSInit

CS'

---

processes' = ∅ ∧ *resources'* = ∅
coordinator'. Grant = ∅ ∧ *coordinator'. Queue* = ∅
communication' = ∅ ∧ *DL_chance'* = *No* ∧ *DL_sure'* = *No*

Now it is shown how various operations executed in the system are specified by operational schemas. All the specified operations in this section include important aspects of concurrent systems:

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 indicates creating a process in the system. In this schema, Δ*CS* expresses the change in the system state (it is considered in other operational schemas similarly). The input process (*p?*) will be created and added to the set of system processes.
To create a process in the system, a number of preconditions must be considered. For example, *p?. PT= Process* shows that this schema just specifies the creation of processes (not threads) in the concurrent system. When all preconditions are satisfied, the new process is added to the collection of all active entities.

DTC

ΔCS
$p?: Pr\_Th;\ new\_t?: \mathbb{P}\ PTName$
$new\_tn!: \mathbb{P}\ PTName;\ new\_create!: \mathbb{P}\ Pr\_Th$

$p?\ .\ Status = Running \land p? \in processes$
$\exists_1 t\_set: \mathbb{P}\ Pr\_Th \bullet \#\ new\_t? = \#\ t\_set$
$(\forall t: t\_set \bullet (t.\ Name \in new\_t?$
$t.\ PT = Thread \land t\ .\ Address = p?.\ Address$
$t.\ Status = Idle \land t \notin processes)) \Rightarrow new\_create! = t\_set$
$new\_tn! = (\mu p: processes \mid p = p?.p.\ ThreadsName \cup new\_t?)$
$processes' = \{\ p: processes \mid p \neq p?\}$
$\cup\ \{\ p: Pr\_Th \mid p\ .\ ThreadsName = new\_tn! \land$
$p.\ Name = p?.\ Name\ \} \cup new\_create!$

DTC specifies dynamic thread creation. 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?*). This operation will cause two changes in the *processes* set: first, the identifier *ThreadsName* of *p?* will change and second, the added threads (i.e., *new_create!*) must be added to all active entities (i.e., *processes*).

Terminate

ΔCS
$p!: \&Pr\_Th$

$\exists p: processes \bullet p \in processes \land p\ .\ Status = Finish \Rightarrow p! = p$
$coordinator'.\ Grant = coordinator.\ Grant \rhd \{p!\}$
$processes' = processes \setminus \{p!\}$

Terminate specifies finishing an active entity (process/thread) in a normal condition. According to the definition of nondeterminism in [9], nondeterministic effects appear in this part of specification since more than one active entity may have the finish status. One of the main reasons to select Z in this work is that the nondeterminism concept has been already added to this language. Thus, the notion of multi-schema (when declaring *p!* by "&") is used according to the notation given in [11].

release

ΔCS
$p?: Pr\_Th;\ r!: \mathbb{P}\ Resource;\ new\_nr!: \mathbb{P}\ Resource$
$new\_tn!: \mathbb{P}\ PTName$

$p? \in processes \land p?.\ Status = Restart$
$r! = coordinator.\ Grant\ ^{\sim}(\{p?\})$
$coordinator'\ .\ Grant = coordinator.\ Grant \rhd \{p?\}$
$new\_nr! = (\mu p: processes \mid p = p? \bullet p.\ NR \cup r!)$
$new\_tn! = (\mu p: processes \mid p = p? \land p.\ PT = Process$
$\bullet\ p\ .\ ThreadsName \setminus p?.\ ThreadsName)$
$processes' = \{\ p: processes \mid p \neq p?\ \} \cup \{\ p: Pr\_Th$
$\mid p\ .\ NR = new\_nr! \land p.\ ThreadsName = new\_tn!$
$p.\ Name = p?.\ Name\ \}$

Release specifies abandonment of all the granted resources to a specific active entity (*p?*) in abnormal conditions (e.g., when an active entity is entered to the *restart* mode). The incidence of abnormal condition will be specified in synchronization schema. In such circumstances, the released resources should be added to the set of active entity requirements (i.e., *NR*).

ReleaseOneResource

ΔCS
$p?: Pr\_Th;\ r?: Resource$

$p?\ .\ Status = Running \land (r?, p?) \in coordinator\ .\ Grant$
$coordinator'.\ Grant = coordinator.\ Grant \setminus \{(r?, p?)\}$

If an active entity does not need its current resource, then releases it. *ReleaseOneResource* specifies liberation of resource (*r?*) that is not needed for the input active entity (*p?*) anymore.

ND_Req

ΔCS
$r?: Resource;\ p!: \&Pr\_Th$

$p! \in coordinator.\ Queue\ r?$
$r? \in resources \mid \mathrm{dom}\ coordinator\ .\ Grant$
$coordinator'.\ Grant = coordinator.\ Grant \cup \{(r?, p!)\}$
$coordinator'\ .\ Queue\ r? = coordinator.\ Queue\ r? \mid \{p!\}$

When several active entities compete for the same resource, nondeterministic effects appear since there may exist more than one active entity which can acquire a specific resource at the same time. Thus, the notion of multi-schema is used for specifying *ND-Req*. In this sense, *ND-Req* is a nondeterministic schema since the result of selecting one of the active entities in a resource queue is uncertain. More precisely, the selection will be done based on different priorities in the implementation phase.
ND-Req is used in two operational schemas, i.e., *Assign_Resource* and *Sinscheduling* as follows:

Assign_Resource

ND_Req
$new\_nr!: \mathbb{P}\ Resource$

$new\_nr! = (\mu p: processes \mid p = p! \bullet p.\ NR \setminus \{r?\})$
$processes' = \{\ p: processes \mid p \neq p!\ \}$
$\cup \{\ p: Pr\_Th \mid p\ .\ NR = new\_nr! \land p.\ Name = p!\ .\ Name\}$

Assign_Resource includes the nondeterministic schema "*ND-Req*" above to complete the specification of resource allocation to an active entity existing in the resource queue.
In the proposed specification framework, two modes of scheduling are specified: *SinScheduling* and *CoScheduling* below are scheduling schemas for single-processor systems and multi-processor systems, respectively:

SinScheduling

ND_Req

$r?.\ Type = Processor \land p!.\ Status = Ready$
$processes' = \{\ p: processes \mid p \neq p!\ \} \cup \{\ p: Pr\_Th$
$\mid p\ .\ Name = p!.\ Name \land p.\ PT = p!\ .\ PT$
$p.\ NR = p!.\ NR \setminus \{r?\} \land p\ .\ EM = p!.\ EM$
$p.\ IM = p!.\ IM \land p.\ Address = p!.\ Address$
$p.\ ThreadsName = p!.\ ThreadsName$
$p.\ Type = p!.\ Type \land p.\ Status = Running$
$p.\ PreviousStatuses = p!.\ PreviousStatuses\ ^{\frown}\langle Ready\rangle\ \}$

As known, several scheduling algorithms exist on a range of different criteria (The logic of these algorithms is used here). Thus, nondeterministic concept is used in this part of the proposed specification framework. *r?.Type = Processor* shows that the input resource must be of type "processor".

According to the definition of "scheduling" in [9], fairness will be guaranteed by a suitable scheduler in the implementation phase, not in the specification stage.
In multi-processor systems, "Coscheduling" should be used because otherwise, active entities will be faced with large communication delays. In the proposed framework, *Gang* scheduler idea [9] is used. According to Gang idea, dependent active entities are executed simultaneously.

CoScheduling

ΔCS
$p?: Pr\_Th;\ r\_set?: \mathbb{P}\ Resource$

$p? \in processes \land p?.Status = Ready$
$\forall r: resources \mid r \in r\_set? \bullet r.Type = Processor$
$r\_set? \subseteq resources \setminus \mathrm{dom}\ coordinator.Grant$
$\#p?.ThreadsName = \#r\_set?$
$\forall r: Resource \mid r \in r\_set?$
$\bullet\ coordinator'.Grant = coordinator.Grant \cup \{(r, p?)\}$
$processes' = \{ p: processes \mid p \neq p? \} \cup \{ p: Pr\_Th$
$\mid p.Name = p?.Name \land p.PT = p?.PT$
$p.NR = p?.NR \setminus r\_set? \land p.EM = p?.EM$
$p.IM = p?.IM \land p.Address = p?.Address$
$p.ThreadsName = p?.ThreadsName$
$p.Type = p?.Type \land p.Status = Running$
$p.PreviousStatuses = p?.PreviousStatuses \frown \langle Ready \rangle \}$

According to the definition of "scheduling" in [9], in *CoScheduling* schema, dependent active entities are gangs scheduled to run simultaneously on distinct processors. Each process consists of a number of interacting threads.
In this part of specification, *r_set?* is an input set of resources, and the type of resources is "processor". This is one of the key preconditions for *CoScheduling* schema. One basic precondition for the Gang scheduler is that the number of *free processors* is equal to the number of dependent threads in *p?* (A *free processor* is a processor that does not belong to any active entity currently). After assignment, the next state of processes (i.e., *processes'*) will be changed similar to *SinScheduling* nearly.

SLS

ΔCS
$r?: Resource;\ hun\_p!: Pr\_Th$

$\exists p: processes \mid p \in coordinator.Queue\ r?$
$\bullet\ p.PreviousStatuses \neq \langle\rangle \land \#p.PreviousStatuses > 1$
$(\forall i: 1..\#p.PreviousStatuses \bullet$
$p.PreviousStatuses\ i = Waiting) \Rightarrow hun\_p! = p$
$processes' = \{ p: processes \mid p \neq hun\_p! \}$
$\cup \{ p: Pr\_Th \mid p.Name = hun\_p!.Name$
$p.PT = hun\_p!.PT \land p.NR = hun\_p!.NR$
$p.EM = hun\_p!.EM \land p.IM = hun\_p!.IM$
$p.Address = hun\_p!.Address$
$p.ThreadsName = hun\_p!.ThreadsName$
$p.Type = hun\_p!.Type \land p.Status = Starvation$
$p.PreviousStatuses = hun\_p!.PreviousStatuses \frown \langle Waiting \rangle \}$

SLS schema specifies Standstill-Livelock-Starvation state based on the definition of "standstill" in [9]. In addition, according to the definition of "starvation" in [9], if all previous statuses of a process are *Waiting*, then the process status is starvation. Meeting several preconditions is necessary to specify "starvation control problem":

- The process (thread) should have some *Previous Statuses* based on which a decision will be made.

$$p.PreviousStatuses \neq \langle\rangle$$

- Minimum number of "*PreviousStatuses"* should be "2". For the first case of *Waiting* mode, the decision is not the *Starvation* mode:

$$\#p.PreviousStatuses > 1$$

- The value of all previous statuses of the input process (thread) for a requested resource should be *Waiting* mode:

$$\forall i: 1..\#p.PreviousStatuses \bullet p.PreviousStatuses\ i = Waiting$$

If the above conditions hold, then the *Starvation* conditions are established and *PreviousStatuses* of the active entity should be changed.

SLI

ΔCS
$p?: Pr\_Th;\ shift\_amount?, length!: \mathbb{N}$

$p? \in processes \land p?.Status = Restart$
$p?.PreviousStatuses \neq \langle\rangle$
$length! = \#p?.PreviousStatuses$
$1 \leqslant shift\_amount? \leqslant length! - 1$
$(length! - shift\_amount? + 1) \bmod 2 = 0$
$p?.PreviousStatuses\ shift\_amount? = Running$
$p?.PreviousStatuses\ length! = Restart$
$\forall i: 1 .. length! \bullet 2 * i - 2 + shift\_amount? \leqslant length! - 1$
$p?.PreviousStatuses\ (2 * i - 2 + shift\_amount?) = Running$
$2 * i - 1 + shift\_amount? \leqslant length!$
$p?.PreviousStatuses\ (2 * i - 1 + shift\_amount?) = Restart$
$processes' = \{ p: processes \mid p \neq p? \} \cup \{ p: Pr\_Th$
$\mid p.Name = p?.Name \land p.PT = p?.PT$
$p.NR = p?.NR \land p.EM = p?.EM$
$p.IM = p?.IM \land p.Address = p?.Address$
$p.ThreadsName = p?.ThreadsName$
$p.Type = p?.Type \land p.Status = InfiniteExe$
$p.PreviousStatuses = p?.PreviousStatuses \frown \langle Restart \rangle \}$

SLI schema specifies Standstill-Livelock-Infinite based on the definition of "standstill" in [9]. Meeting several preconditions is necessary to specify "Infinite execution":

- The current state of the input process (thread) should be *Restart*.
- Considering a shift number (*shift_amount?*) on the sequence of *PreviousStatuses* as a starting point.
- Calculating the total length (*length!*) of the *PreviousStatuses* sequence.
- Having a *Running* condition in the *shift_amount?* position on *PreviousStatuses* sequence.
- Having a *Restart* condition in the *length!* position on *PreviousStatuses* sequence.

Continuously switch between the two conditions *Restart* and *Running* from determined position of *shift_amount?* to determine the position of *length!* .
Now according to the definition of "standstill" in [9], livelock situation is specified as follows:

LiveLock

SLS
SLI

CircularCondition

ΔCS
$p?: Pr\_Th;\ r?: Resource;\ len\_set!: \mathbb{N}$

$r? \in resources$

$\exists p\_set$: 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 active entities. The output of this schema is either *Yes* or *No*. *p?* and *r?* are input active entity and resource, respectively. *p?* intends to access *r?*. This access can cause deadlock in the system under certain conditions.

Synchronization
CircularCondition

p? ∈ *processes*
p?. Status ∉{*Finish*, *Waiting*, *Restart*} ∧ *r?* ∈ *p? . NR*
r? ∈ *resources* \ dom *coordinator . Grant*
⇒ *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*
⇒ *DA = DetRec* ∧ *DL_chance* ∈{*Yes*, *No*} ∨
DA = AVO ∧ *DL_chance = No*
⇒ *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* ⇒ *DL_sure' = Yes*
DA = AVO ∧ *DL_chance = Yes* ⇒ *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 the definition of "synchronization" in [9], active entities need to be synchronized. In *Synchronization* schema, synchronization is done based on two types of deadlock approaches. *CircularCondition* schema has been also used in this schema. *p?.Status* ∉ {*Finish*, *Waiting*, *Restart*} shows unauthorized statuses for *p?*.
As mentioned, if the value of *DL_chance* is *Yes* in Detection & Recovery approach, then deadlock will occur definitely. *Deadlock_Recovery* specifies deadlock problem elimination:

DeadLock_Recovery
Synchronization
p_loop?: ℙ *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*

If the deadlock approach is Detection & Recovery, then it is resolved by killing a process or thread existing in the detected cycle randomly (or nondeterministically); hence, the notion of multi-schema was used when specifying *DeadLock_Recovery*. *p_loop?* shows a set of involved active entities in the loop of deadlock, and *p!* shows a nondeterministic representative of *p_loop?*. Finally, when the loop is not present, the value of *DL_sure* will be *No*.
According to the definition of "communication" in [9], active entities can be communicated in two methods: by *message passing* or *shared variables*. Communication by message passing can be either *synchronous* or *asynchronous.*

Asynchronous_Communication
Δ*CS*
p?: Pr_Th; r?: Resource; new_M!: ℙ *Message*

CT = MessagePassing ∧ *r? . Type = Network*
(*r?*, *p?*)∈ *coordinator . Grant*

In the asynchronous message passing, a message can be placed on a network location, provided that 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 operations of sending and receiving messages, respectively:

As_Send_Me
Asynchronous_Communication
m?: Message

∃*q: processes* • (*p?*, *q*)∈ *communication*
m? ∈ *p? . EM* ∧ (*r?*, *p?*)∈ *coordinator . Grant*
new_M! = (μ*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?*⟩}

As_Receive_Me
Asynchronous_Communication
m!: Message

∃*q: processes* • (*q*, *p?*)∈ *communication*
r?. Location ≠⟨⟩ ∧ *m! = head r?. Location*
new_M! = (μ*p: processes* | *p* = *p?* • *p. IM* ∪{*m!*})
processes' = { *p: processes* | *p* ≠ *p?* } ∪

$\{\ p: Pr\_Th \mid p\ .\ IM = new\_M! \wedge p.\ Name = p?\ .\ \ Name\}$
$resources' = \{\ r: resources \mid r \neq r?\ \} \cup \{\ r: Resource$
$\mid r\ .\ Type = r?.\ Type \wedge r.\ Location = tail\ r?.\ Location\ \}$

As it is clear from the name of *"synchronous communication"*, "send" and "receive" operations are done simultaneously:

Synchronous_SeAndRe

ΔCS
$p?, q?: Pr\_Th;\ m?: Message;$
$new\_pm!, new\_qm!: \mathbb{P}\ Message$

$CT = MessagePassing \wedge (p?, q?) \in communication$
$m? \in p?\ .\ EM$
$new\_pm! = (\mu p: processes \mid p = p? \bullet p\ .\ EM \setminus \{m?\})$
$new\_qm! = (\mu q: processes \mid q = q? \bullet q\ .\ IM \cup \{m?\})$
$processes' = \{\ p: processes \mid p \neq p? \wedge p \neq q?\ \} \cup$
$\{p: Pr\_Th \mid p\ .\ EM = new\_pm! \wedge p\ .\ Name = p?.\ \ Name\}$
$\cup\ \{q: processes \mid q.\ IM = new\_qm! \wedge q.Name = q?.\ \ Name\}$

Synchronous_SeAndRe schema specifies synchronous message passing. According to the definition of "communication" in [9], 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
$p?: Pr\_Th;\ r?: Resource;\ new\_M!: \mathbb{P}\ Message$
$new\_r!: Resource$

$CT = SharedVariable \wedge r?\ .\ Type = Memory$
$(r?, p?) \in coordinator\ .\ Grant$

Write_Message

$SharedMemory\_Communication$
$m?: Message$

$\exists q: processes \bullet (p?, q) \in communication$
$m? \in p?\ .\ EM$
$new\_M! = (\mu p: processes \mid p = p? \bullet p\ .\ EM \setminus \{m?\})$
$processes' = \{\ p: processes \mid p \neq p?\ \} \cup \{\ p: Pr\_Th \mid$
$p\ .\ EM = new\_M! \wedge p.\ Name = p?\ .\ Name\ \}$
$resources' = \{\ r: resources \mid r \neq r?\ \} \cup \{\ r: Resource$
$\mid r\ .\ Type = r?.\ Type \wedge r.\ Location = r?.\ Location \frown \langle m? \rangle\ \}$

Read_Message

$SharedMemory\_Communication$
$m!: Message$

$\exists q: processes \bullet (q, p?) \in communication$
$r?.\ Location \neq \langle\rangle \wedge m! = head\ r?.\ Location$
$new\_M! = (\mu p: processes \mid p = p? \bullet p.\ IM \cup \{m!\})$
$processes' = \{\ p: processes \mid p \neq p?\ \}$
$\cup \{\ p: Pr\_Th \mid p\ .\ IM = new\_M! \wedge p.\ Name = p?.\ \ Name\}$
$resources' = \{\ r: resources \mid r \neq r?\ \} \cup \{\ r: Resource$
$\mid r\ .\ Type = r?.\ Type \wedge r.\ Location = tail\ r?.\ Location\ \}$

According to the definition of "communication" in [9], in shared memory systems, processes/threads communicate with each other using two operations write and read on shared variables; these operations are similar to send and receive in the asynchronous communication.
The next section investigates comprehensiveness of the presented framework. It also discusses on the appropriateness of Z to being used for formal specification of concurrent systems.

4. EVALUATION OF THE PRESENTED FRAMEWORK

As mentioned in section 3, the final specification has been type checked and consistency checked using a well-known Z type checker, i.e., Z/eves 2.1. Thus, the presented specification in this paper is structured correctly. To evaluate the proposed specification framework in terms of comprehensiveness, Table 2 is presented. In this table, the proposed framework is compared with the previous related work; see section 2 for reviewing related work.
The second column of table 2 shows the results of previous work and related shortcomings. The third column implies the comprehensiveness of the framework presented in this paper in comparison to related work. In several parts of the fourth column, "making nondeterminism explicit" was considered as one of the advantages of the proposed framework because as the findings of [11] show, when nondeterministic behavior of concurrent systems is specified *explicitly*, all interleaved executions of concurrent processes will be extractable in the final program. In other words, it will be possible to develop real, concurrent systems formally; see [4] and [11] for more details.

Table 2. Comparing specified aspects in the proposed framework with other related work

Aspect	Related work	The proposed specification framework	Advantages of the proposed specification framework
Dynamic Process / Thread Creation	Lack of a suitable specification	Thread creation dynamically: create and terminate active entity	Multi-threading specification; Suitable specification for concurrency concept; Making nondeterminism explicit
Communication	Only message passing specification	Both Message passing and Shared variable specification	Comprehensive specification of the communication
Synchronization	Specification of classical case studies	Precise specification of issues related to assign resources	Conjunction with the approach to deal with deadlock implicitly; Making nondeterminism explicit

Standstill (Deadlock & live lock)	Only deadlock detection specification	Comprehensive specification of deadlock and livelock	Making nondeterminism explicit; Appropriately descriptive approach
Scheduling	Lack of a suitable specification for concurrent systems	Specification of scheduling in single processor systems and using the Gang scheduler idea to specify scheduling in multi-processor systems	Scheduling specification in single processor and multi-processor systems; No delay in communication; Making nondeterminism explicit in single processor systems

As another benefit of the proposed work, it should be reemphasized that the work is based on a single formalism, i.e., Z. The following samples are extracted from Table 1 in order to show how various formalisms have been used in previous work in order to specify some aspects of concurrent systems:

1. Synchronization specification: Z and Petri Net [19]
2. Deadlock detection specification: Z and Temporal Logic [8]
3. Buffer case study specification: Circus (Z and CSP) [17]

Unlike cases 1 and 2 above, in the proposed framework, synchronization and deadlock detection are specified only using the Z notation: since nondeterminism is specified in this framework explicitly, the Z notation can be used independently. In the specification of buffer (case 3 above), there are synchronization and communication concepts implicitly. In this case study, a combination of Z and CSP has been used since Z was unable to specify the dynamic behavior of concurrent systems at that time. In the following, it is shown how one can use the framework of this paper to specify operations which are equivalent with those operations existing in the buffer case study, i.e., input and output operations (see [17] for more details):

Input

Synchronization
Write_Message

Output

Read_Message

In this way, the presented framework is able to specify behavioural and functional aspects of the buffer case study only based on the Z notation. Moreover, due to existence of *synchronization* in *input* schema, one important issue, i.e., assigning resources, has also been considered here. This issue is not considered in [17].

In summary, the presented specification framework is only dependent on a single notation, i.e., Z. This feature releases us from integration efforts to have a comprehensive specification of concurrent systems. Also, several existing methods for formal verification and formal program development, which rely on Z specifications, can be applied to the provided specification.

5. FUTURE WORK

One goal to specify systems formally is developing programs from formal specifications. Z has been chosen since it has an interpretation in Martin-Löf's theory of types [11]. Thus, as a future work, the authors are going to translate the Z specification of a concurrent system 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, it will be possible to provide a completely formal way to specify and develop concurrent systems.

6. REFERENCES

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

[2] A. J. Bijoy, D. P. Hiren, "Generating Multi-Threaded Code from Polychronous Specifications", ElsevierJournal, ENTCS, vol. 238, 2009, 57–69.

[3] 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.

[4] H. Haghighi, "Towards a Formal Framework for Developing Concurrent Programs: Modeling Dynamic Behavior", Proc. AICCSA-10, Tunisia, 2010.

[5] S. C. Harpreet, W. B. John, M. W. Jeanette, "Formal Specification of Concurrent Systems", Advances In Engineering Software, vol. 30, 1999, 211–224.

[6] D. Safranek, "Visual Specification of Systems with Heterogeneous Coordination Models", ENTCS, 2007 107-121.

[7] P. Stocks, K. Raymond, D. Carrington, A. Lister, "Modeling Open Distributed Systems in Z", Elsevier computer Communications, vol. 15, 1992, 103–113.

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

[9] S. Sharifirad, H. Haghighi, "A Comprehensive and Integrated Framework for Formal Specification of Concurrent Systems", International conference on software engineering and technology, Venice, Italy, November 2011.

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

[11] H. Haghighi, S. H. Mirian-Hosseinabadi, "Nondeterminism in Constructive Z", Fundamenta Informatica, vol. 88, 2008, 109-134.

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

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

[14] J. C. P.Woodcook, C. Morgan, "Refinement of State-Based Concurrent Systems", Proc. VDM 90, Springer Verlag, 1990, 341–351.

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

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

[17] J. C. P. Woodcock, A. Cavalcanti, "A Concurrent Language for Refinement", Irish Workshop in Formal Methods, 2001, 1–16.

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

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

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

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

[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, 1990, 133–148.