Modelling the semantics of multitasking facilities in Concurrent C using Petri nets

Abdulaziz Boujarwah*, Nadia Al-Seif, Kassem Saleh

Kuwait University, Department of Electrical and Computer Engineering, P. O. Box 5969, Safat 13060, Kuwait

Abstract

Concurrent C (ConcC) [1] is a relatively new language that extends the C programming language [2] by incorporating multitasking facilities. These facilities are mainly based on the bidirectional rendez-vous concept, and they offer flexible and efficient mechanisms for the specification of concurrency, communication and synchronization often required in real-time systems. However, as for most real-time and concurrent programming languages such as Ada [3], the description of multitasking facilities in ConcC is lacking a rigorous and formal semantical definition. Most existing introductory materials to ConcC by its authors are informally described, and cannot be easily understood. In this paper, we explain the semantics of multitasking facilities in ConcC using the Petri net model. As is well known, the Petri net model is a formal graphical model that can describe concurrent behaviours of systems in a natural and intuitive manner, and yet it is based on strong mathematical and semantical foundations. Timed, inhibitor arc and predicate Petri nets are extensions that are used to enhance the expressive power of the basic Petri net to model and explain all ConcC multitasking features.

Keywords: Concurrent C; Concurrent programming; Multitasking; Petri nets; Real-time systems; Semantics

1. Introduction

The fast growing area of real-time distributed systems applications necessitated the introduction of new high level programming languages suitable for the implementation of distributed systems paradigms such as concurrency, communication and synchronization. Various new languages were defined to deal with such advanced requirements and concepts, among which we find Ada, Modula, CSP and many other languages [4]. Also, existing languages were extended to include the syntactic and semantic support of such new paradigms. Among such languages, we find Concurrent C and Concurrent Pascal [4,5]. This paper deals specifically with Concurrent C.

Concurrent C provides an extension of the C programming language by incorporating multitasking facilities. These facilities are mainly based on the bidirectional rendez-vous concept. However, there exist major differences between Ada's and Concurrent C's (ConcC) rendez-vous semantics that affect the timeliness of the implementation of real-time applications to the advantage of ConcC. For example, in an Ada Task, messages arriving from other processes are serviced in a strict FIFO order, whereas, in ConcC, the receiving task can optionally service the (high priority) message incoming from a high priority process by fetching the queue. Also, in Ada, all predicates associated with a task entry point are evaluated before a non-deterministic choice is made to accept one of the entry points associated with a true predicate, and a rendez-vous occurs. A complete comparison between the semantics of multitasking of Ada and ConcC is provided later in the paper. Similarly to Ada, the description of multitasking facilities in ConcC is lacking a rigorous and formal semantic definition [6]. Most existing introductory materials to ConcC by its authors are informally described, and cannot be easily understood.

In this paper, we explain the semantics of multitasking facilities in ConcC using the Petri net model. As is well known, the Petri net model, unlike other models such as process algebra and temporal logic models, is a formal graphical model that can describe concurrent behaviours of systems in a natural and intuitive manner, and yet it is based on strong mathematical and semantical foundations. Timed, inhibitor arc and predicate Petri nets are extensions that are used to enhance the expressive power of the basic Petri net to model and explain all ConcC multitasking features.

The rest of this paper is organized as follows. The next section provides some preliminary background to the Petri net model and its variations, and introduces the basic concepts for interprocess communication and multitasking in high level programming languages. The following section describes the semantics of each of the multitasking facilities of Concurrent C using the Petri net model. Summaries of the multitasking features of Concurrent C follow, with conclusions in the final section.

2. Preliminaries

We first provide a formal definition of the basic Petri net...
model and its variations that are used in this paper. Then, we introduce some basic concepts for multitasking in high level programming languages such as the rendez-vous concept and interprocess communication and synchronisation.

The Petri net model and its variations

The Petri net model is a powerful model used for the description of concurrent systems' behaviours in an intuitive manner, yet it has strong semantics. The model was named after C. A. Petri [7] who first introduced it. Real-time systems, operating systems and computer programs were modelled, among other systems, using Petri nets. In this paper, we use Petri nets to model and explain the semantics of multitasking facilities in Concurrent C. However, the verification aspect of Petri net modelled systems is beyond the scope of our interest. The interested reader can refer to Peterson [8]. For the purpose of this paper, we only introduce the semantics and the descriptive power of the model, and how it can be used to explain multitasking in Concurrent C.

A basic Petri net, or simply a Petri net (PN), is a quintuple PN(P,T,B,F,M), where: P is a non-empty set of places and card(P) = p, T is a non-empty set of transitions and card(T) = t, B: P × T → N, is a backward incidence function which determines the existence of arcs emanating from places to transitions in the net, F: P × T → N, is a forward incidence function which determines the existence of arcs emanating from transitions to places in the net, and finally, M0 is the initial marking of the net which determines the initial distribution of tokens inside the different places of the net.

A PN can be represented graphically as follows: (1) a place is represented by a circle; (2) a transition is represented by a horizontal bar; (3) a forward (backward) arc is represented by an arc starting from a place (transition) and terminating at a transition (place); and finally, (4) a token is represented by a bold dot, and i tokens exist inside a place p if M0(p) = i. The initial distribution of tokens, in M0, among the many places of the net is an issue that depends on the initial state of the system to model. Moreover, a token could be either a primitive token undistinguished from any other token, or a non-primitive or structured token which may hold identifying information.

Given a marking M, we say that a transition t of a PN is enabled when all the following conditions are satisfied: (1) no tokens exist in place p1; (2) at least one token exists in place p2; and (3) predicate p evaluates to true. Once t is enabled, it must fire after the elapse of tMIN but before the elapse of tMAX. The tokens are removed from the pre-condition places only when the timed transition is fired. TPNs are very useful tools for the modelling and verification of real-time properties.

For example, in the PN shown in Fig. 1, transition t will be enabled when all the following conditions are satisfied: (1) no tokens exist in place p1; (2) at least one token exists in place p2; and (3) predicate p evaluates to true. Once t is enabled, it must fire after the elapse of 2 time units and before the elapse of 5 time units. Once t is fired, one token is removed from p2 and one token is gained in each of the output places.

Petri nets have been used to model, among other things, sequential as well as parallel computer programs. Fig. 2 shows both the pseudocode of a concurrent program and its equivalent PN.

Concepts and paradigms for multitasking

In a real-time and concurrent system or environments, the major players are the communicating entities often called...
processes or tasks. In such systems and in a client-server architecture, entities may require to rely on each other to solve certain problems or subproblems. Therefore, some mechanisms for inter-entity communications must be provided, at a high level, by the real-time language. The necessary mechanisms include the support for communication, synchronization and basic concurrency. The syntactic and semantic support for the specification of concurrent entities has been supported differently in the existing real-time languages. For example, tasks definitions and bodies are in Ada, process declaration and definition are in ConcC, and monitors are in Concurrent Pascal, Concurrent Euclid and Modula. Also, communication between concurrent entities is supported by these languages. For example, in Ada and ConcC, communicating entities (tasks and processes) can interact through well-defined interfaces (i.e. the entry points), and in Concurrent Pascal, by using procedure entry call mechanism. Finally, to achieve communication, communicating entities have to synchronize and mutually agree on the exchange of information.

The synchronization among two entities for the purpose of communication is known as a rendez-vous. In the rendez-vous approach, two processes have to synchronize and agree on the communication for the exchange of information to take place. Processes come together and exchange information, after which they proceed with their executions independently. The flow of exchanged information could be either unidirectional or bidirectional. It is clear from the above description of the rendez-vous semantics that it involves some delays due to the possible waiting of one process for other processes: caller or callee. Both ConcC and Ada have a queue associated with each entry point. A record is placed in the queue when a process (or task) performs a transaction (or entry) call.

3. Multitasking in Concurrent C

ConC uses the bidirectional rendez-vous or transaction concept to allow a bidirectional information transfer during the rendez-vous [12]. The process that executes a transaction call (i.e. client process) requesting a service from another process is automatically forced to wait until the called process (i.e. server process) is ready to execute the request. The client process can continue its execution only after the transaction results are returned.

ConC provides the syntactic constructs for supporting the mechanisms for multitasking and rendez-vous. These facilities include: (1) declaration and creation of processes; (2) process synchronization and interaction; (3) process termination and abortion; (4) process parameterization; and (5) priority specification.

A ConC process is able to receive transaction calls from other processes and executes them using the select statement. Also, a process is able to make a transaction call to other processes. A process can play the dual role of a client and a server process. A server process must have designated and typed entry points. A queue is associated with every transaction entry point in which transaction calls are saved before being processed in a rendez-vous. However, the retrieval or servicing of requests from the queue is not strictly FIFO, since a process can optionally fetch the queue for a request coming from a high priority process or for a request satisfying certain selection criteria.

The select statement

The select statement allows a process to select one alternative (one or more statements) among several alternatives separated by or. An alternative can be an accept alternative, an immediate alternative, a delay alternative, or a terminate alternative.

A conditional execution of a select alternative can be introduced using a guard. A guard is a boolean expression which indicates the condition that must be true for an alternative to be a candidate for execution. The guards are evaluated in their textual order of appearance in the select statement. The accept alternative has the highest priority over all alternatives, then the immediate alternative, delay alternative, and finally the terminate alternative.

ConC allows transactions to be deterministically accepted in a user-specified order. Therefore, it is the programmer’s responsibility to order the different alternatives in the select statement. The priority within the accept alternatives is determined by the order of appearance in the select statement. By default, transaction calls are accepted in first in first-out (FIFO) order for different client processes in the...
queue. The by clause of the accept statement allows the acceptance order to be altered in the queue. On the other hand, the suchthat clause allows the selection of the transactions call to be based on the arguments of the transactions in the queue.

Accept alternatives

As mentioned above, the accept alternative has the highest priority among other alternatives in the select statement. Normally, accept statements appear at the beginning of the select statement and can be either guarded or non-guarded.

Non-guarded accept. Fig. 3 shows a part of the Petri net describing a non-guarded accept statement. The first accept is chosen by firing the transition 'enable' that produces a token in the place 'open'. To engage in a rendez-vous at the accept statement, there must exist one outstanding transaction call. The place 'queue' is an external place that holds a token for each outstanding transaction call made to the process. These tokens are structured and each one of them holds information related to an individual transaction call. The transaction call (i.e. token) with the least by clause value in the place 'queue' (optional), and satisfying the suchthat clause (optional) will be selected. In this case, the transition 'StartRV' can fire, and the statements after the accept are executed (i.e. the process is engaged in the rendez-vous). However, if one of the above conditions is not satisfied, transition 'next' is fired and the next alternative in the select statement is considered. Once the process engages in the rendez-vous, it performs the corresponding statements in the accepted entry and returns to the beginning of the select statement again.

Guarded accept alternative. Fig. 4 shows the semantics of a guarded accept alternative. The meaning of the place 'queue' is explained above. Transition 'StartRV' will be enabled when the guard associated with the transition is evaluated to true. Consequently, the same conditions as for non-guarded alternatives will be applied. Otherwise, a token is gained in the place 'close', and the guard of the next alternative will be evaluated, and so on. Furthermore, if all the alternatives of the select statement have guards and all are evaluated to false, that is each place 'close' contains a token, the transition 'error' will fire and an error condition will be raised. The user will be notified at run time.

Immediate alternative

Normally, an immediate alternative should appear textually after all accept alternatives. If no rendez-vous is possible with an accept alternative, the immediate alternative will be considered. Immediate alternatives can be either guarded or non-guarded. Fig. 5 shows the case of a guarded immediate alternative. If the guard is evaluated to true, then the actions associated with the alternative will be executed.

![Fig. 3. Non-guarded accept alternative.](image1.png)

![Fig. 4. Guarded accept alternative.](image2.png)
Fig. 5. Guarded immediate alternative.

**Delay alternative**

Textually, the *delay* alternative should appear after the *immediate* alternative. Similarly, the *delay* alternative can be either guarded or non-guarded. The non-guarded *delay* alternative is also called open delay. Fig. 6 shows the select statement with guarded *delay* alternatives. All the guards for the *delay* alternatives are evaluated first, and the minimum delay with a true guard is chosen. When the *delay* alternative is open, and the delay timer starts, any arrival of a call with a true guard will interrupt the delay, and control will go back to the *select* statement. Otherwise when the delay timer expires and all the queues for the transaction calls are empty (i.e. the external place 'call arrived' has no tokens), the statements after the delay are executed non-interruptibly.

The situation is simpler if the delay does not have guard (open delay), then the places and transitions related to the guard are removed from Fig. 6, and the delay is executed directly without comparing it with other delays if any exist. Moreover, an open *delay* or an open *immediate* alternative cannot exist with the *terminate* alternative, since they have a higher priority over the terminate. Therefore, they will always be executed in this situation.

**Terminate alternative**

The *terminate* alternative has the lowest priority over all *select* alternatives. Fig. 7 shows the *select* statement with a *terminate* as a last alternative. A guarded *terminate* alternative cannot be selected unless the following conditions are satisfied:

1. The guard is evaluated to true.

Fig. 6. Guarded delay alternative.

Fig. 7. Guarded terminate alternative.
(2) There is no outstanding transaction call in all other queues.
(3) All other processes have completed, terminated, or are waiting at a terminate alternative, i.e. the master and (td1,...,tdk) dependent processes.

The figure contains an external place to contain a token only when the master and all dependent processes have terminated or waiting at a terminate alternative. Also, another external place is used to contain a token if there is at least one outstanding transaction call at an entry point of the process. Note that when the choice is between a terminate alternative that cannot be selected immediately and a delay alternative with true guards, then the result is undefined in the sense that the select statement semantics does not state which alternative will be taken.

There is a more powerful explicit termination statement, c-abort, which unconditionally and immediately terminates a process. A process can abort itself or other processes.

Priority specification

Process priorities in ConcC are specified explicitly by a parent process when creating a child process. However, process priority can be changed at run-time. Facilities for specifying and modifying priorities are provided. Moreover, different instances of the same process type can be assigned different priorities explicitly using:

create process-type-name(initial-values) [priority(p)]

If the priority is omitted, then the new process is assigned a standard (default) priority. A process can change its own priority; however, the priority can also be changed by another process using some standard library functions.

More on the select semantics

In the following, we explain two additional semantics associated with the above facilities and constructs.

Clearing close places. Places labelled 'guard' may contain tokens when the guard is evaluated to true. However, if the suchthat clause is false or the queue is empty, the 'guard' place will not release its tokens. On the other hand, if the guard is evaluated to false, transition 'false guard' will fire a token in the place 'close', this place must be emptied for the next execution of the select statement if one select alternative has been chosen. The modifications shown in Fig. 8 introduce a new output place 'clear' from each transition marked with an asterisk, connected to each pair of 'guard' and 'close' places for each of the previous guarded select alternatives. Thus, when any alternative is chosen, it will clear all the previous places that contain tokens.

Timed transaction call. A timed transaction call statement allows a process to withdraw a transaction call if it is not accepted within a specific period, using the following statement:

within (specified-duration)?

process. (transaction_name) (actual-parameters): (expression)

where: (process) is a valued expression that returns a value as a timed call expression and (expression) is an expression that is evaluated only when the transaction call is withdrawn, and the expression value becomes the timed call expression. Fig. 9 describes the timed transaction calls mechanism.

4. Summary of the multitasking features of Concurrent C

In this section, we briefly summarize the multitasking
features of ConcC and their impact with respect to the flexibility and the responsiveness or timeliness of ConcC programs. These features are listed below:

1. In ConcC, transaction calls are accepted in FIFO (queue) order by default. But the by clause can change this order. This adds more flexibility in the modelling and problem-solving power of ConcC.

2. In ConcC, the select statement may contain one or more (guarded or non-guarded) accept alternatives, immediate alternatives, delay alternatives or a terminate alternative. The accept statement has the highest priority, then the immediate, the delay, and finally the terminate alternative.

3. In ConcC, if there is more than one accept statement in the select statement, the first ‘open’ accept statement is deterministically chosen. Therefore, the lexical order of the accept statements is important for prioritizing the order of execution of open statements. This allows an unambiguous specification and modelling of real life systems.

4. In ConcC, a process is created explicitly using ConcC library functions. The priority of a process can be specified and changed explicitly and dynamically. Therefore, allowing the user a greater control over the creation of tasks.

5. In ConcC, when the accept alternative with a true guard is chosen, then there are two (optional) conditions that must be satisfied for the alternative to be executed:
   (a) The suchthat clause should be true.
   (b) It should have the minimum by clause value in the queue.

6. In a timed transition call in ConcC, there is an expression that is evaluated and returned as a value to the timed call expression when the process duration time expires and the process is withdrawn.

7. In ConcC, a delay statement can be interrupted (i.e. by the arrival of a transaction call). Therefore, enhancing the timeliness (overall response time) of the software system.

In general, the semantics of multitasking facilities of ConcC offer an efficient handling of concurrency with respect to response time. Specifically, items (3) and (7) allow fast implementations of interprocess communication mechanisms. Moreover, ConcC is adequately flexible in allowing both FIFO and non-FIFO management in servicing the incoming messages to a process (items (1) and (5)).

5. Conclusions

Concurrent C is a real-time programming language that extends the C programming language by offering some additional syntactic constructs to deal with concurrent programming paradigms. These syntactic constructs have powerful semantics that cannot be described precisely or explained in a natural language such as English. In this paper, we explain the semantics of concurrency, communication and synchronization in Concurrent C using a powerful graphical model for describing concurrency, namely, the Petri net model. Extensions to the Petri net are used to model the semantics of ConcC. For example, Predicates Petri nets are used to explain the semantics of the guarded alternatives, such as guarded accept and guarded delay and terminate statements. Similarly, Timed Petri nets were used to model the different timeout mechanisms associated with some of the alternatives. Finally, Inhibitorarcs were used to model situations where firing is based on the unavailability of tokens in incident places.

Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments which helped to improve this paper. We would like also to acknowledge the support of this work by Kuwait University Research Grant EE-059.

References