When Concurrent Control Meets Functional Requirements, or Z + Petri-nets*

Frédéric Peschanski[1] and David Julien[2]

[1] University of Tokyo pesch@yl.is.s.u-tokyo.ac.jp
[2] Laboratoire d'Informatique de Paris 6 David.Julien@lip6.fr

Abstract. It is our belief that the formal design of real-world concurrent systems does not fit well with model/state-oriented specification languages such as the Z notation. The problem with such systems is that they not only expose complex functional requirements but also critical control-level aspects such as concurrency. On the other hand, the most widely-spread formal languages dealing with concurrency, namely Petri-nets, reveal weaknesses (mostly state-space explosion) when dealing with complex functional requirements. In this paper, we propose a hybrid methodology, based on the traditional Z notation for the functional part of the system and using Petri-nets to model its concurrent control. We describe a simple method to derive new proof obligations in case of possible concurrent activation of Z operations, as modeled by the associated Petri-nets. By keeping the interface between both worlds as thin as possible, we do not put into question the interesting properties of the Z language : expressiveness, modularity and support for refinement. Moreover, our petri-based concurrent activation networks only address concurrency issues. Hence, it is likely that they remain manageable in term of state-space and so analyzable using existing Petri-net tools. We experimented this exploratory method on a real application, a research middleware kernel, which is now fully operational.

1 Introduction

The Z notation [11] supports the formal specification of computer systems with great freedom thanks to its modular and open design. However, most of Z-based industrial-strength specifications have been done on sequential systems, using models of states and transitions of the systems. Our research interest is focused on the design and implementation of concurrent and distributed systems and languages. Our main project is a component-based middleware system called Comet [8]. The problem with such systems is that they expose not only various and complex functional requirements but also critical control-level aspects. These aspects —we will focus on *concurrency* in this paper— do not fit well with standard state/transition models. As a matter of fact, we think that traditional Z-based methodologies fail to both (1) provide expressive abstractions to model

* This work is supported by the Japan Society for the Promotion of Science (JSPS) under research grand #14-02748.

concurrent control and (2) guide system designers regarding proof obligations when concurrency is involved.

Petri-nets and related formal languages [10], on the contrary, provide abstractions that are generally considered as very simple (denotationally speaking) and expressive to model concurrent systems. Formal techniques and support tools allow the relatively easy (and often automated) conduction of proof regarding concurrency such as finding deadlocks or starvation conditions and so on. However, despite the important research advances in the domain [4], it seems still difficult to model and analyze large specifications with complex functional requirements using only Petri net-based languages because of the large state-space these denote.

Therefore, we propose in this paper a hybrid methodology employing Z schemas to fulfill the functional requirements of the systems and *concurrent activation nets*, based on Petri-nets, to model concurrency as well as deriving new proof obligations at the functional level.

Sect. 2 of the paper presents our case study : the kernel of an operational research middleware platform. We present some extracts of the Z-based functional specification of this system. We then describe in Sect. 3 the issues raised by the concurrent nature of the case study. To address these issues, we model using Petri-nets a set of concurrent activation networks for the related functional specifications. We also explain our method to derive, from the activation nets, the new proof obligations for the specification designers. We design in Sect. 4 an extension of our middleware kernel showing that Z-based modular specifications and refinement techniques are not compromised by our approach. We then overview some related work and conclude the paper.

2 Case-study : The Comet kernel

The Comet middleware is a research experiment attempting to support highly dynamic component-based distributed systems based on an asynchronous model of communication [9]. One advantage of such asynchronous form of communication is its implicit concurrent nature resulting in a great autonomy of the involved components in term of control. The second most fundamental property of the system is that the deployed components do not make explicit reference to other components, they can only be related at runtime using connections. They then can exchange information in the form of asynchronous emissions of events. We thus enforce in Comet an explicit coupling relation among components, both in structural (explicit references) and behavioral (asynchronous control) terms so that the systems can be built dynamically and incrementally. In this paper, we are mainly interested by the control-level aspects —mostly concurrency— of the component internals.

From the previous discussion we can state that a component, in our middleware, is basically an *asynchronous software entity*. As in other concurrent models (such as actor languages [1] or active objects [2]), this implies a decoupling between the communication system and the execution layer. In our event-based approach, we distinguish :

– the reception of the events by a given component,
– the corresponding execution stage, and
– the potential resulting emission of output events.

Fig. 1 shows the various functional requirements of our asynchronous management of events within component boundaries. First, the *Receive* and *Send* functionalities explain the links between the components and the underlying communication system. Events, when received, are queued in the *InQueue* structure and then fetched concurrently by the *InFetch* operation. The execution phase is handled by *Exec* and output events are emitted asynchronously through *OutQueue*. The *OutFetch* phase is also decoupled in term of control from the other phases (except *Send*). We also define other structures and operations to describe the communication layer as well as other higher-level considerations (e.g. connection models).

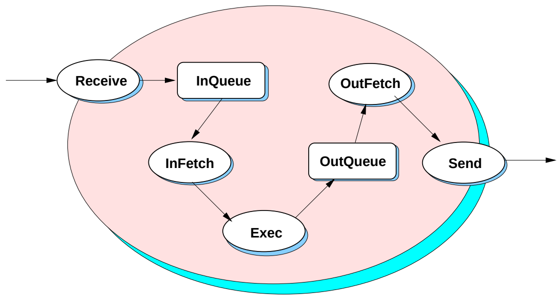

Fig. 1. Functional requirements (operation schemas)

All these structures and operations, namely the *functional requirements* of the system, have been specified and refined using the Z notation. The whole abstract (i.e. not refined) specification is composed of about one hundred schemas, resulting in a middle-sized model. Of course, we will only present a small subsets of these schemas, considering only (and partially) the operations involved within component boundaries.

2.1 Component state

The state manipulated within a component is composed of a set of input and output channels as well as two queuing systems. The corresponding Z-schema is given below:

$$
\begin{array}{|l}
\hline
\textit{Behavior} \\
\hline
in\_channels : \mathbb{P}\ CommChannel \\
out\_channels : \mathbb{P}\ CommChannel \\
in\_queue : \mathrm{seq}\ Event \\
out\_queue : \mathrm{seq}\ Event \\
local\_id : \mathbb{N} \\
\hline
local\_id \geq 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{CommChannel} \\
\hline
id : IDENTIFIER \\
type : \mathbb{P}\ TYPE \\
events : \mathrm{seq}\ Event \\
\hline
\end{array}
$$

To remain concise, we don't show here the whole state and invariant, pointing only the essential aspects for event management. The only condition we expose is that the *local_id* counter should be positive[3]. This counter ensures that currently processed events have different local identifications. This invariant condition is not expressed exactly under these terms in our real system but this will be helpful to show some issues regarding concurrent activation later on. The *CommChannel* structure corresponds to an event channel between two components. It mainly exposes a $\mathrm{seq}\ Event$ data, that is, a sequence of events actually exchanged among the components. The *in_queue* and *out_queue* are also represented as sequences of events.

We will suppose that the component state is initially empty so that all sequences are empty ($\langle\rangle$) and the local identifier is zero. Another hypothesis is that some input and output channels have been set up in the initial configuration so that we don't have to go into the details of the complex establishment and removal of communication channels.

2.2 Operations

The reception of an event is a two-steps process. First, a particular input channel conveying new events must be chosen. This can be done by fetching an element c of *in_channels* non-deterministically, under the condition that $c.events \neq \langle\rangle$. Then, we have to extract one event from the channel and output it. The emission phase will consist in the converse operation (pushing an event into a channel). The *Receive* and *Send* operations are defined below:

$$
\begin{array}{|l}
\hline
\textit{Receive} \\
\hline
\Delta CommChannel \\
ev! : Event \\
\hline
events \neq \langle\rangle \\
ev! = last\ events \\
events' = front\ events \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Send} \\
\hline
\Delta CommChannel \\
ev? : Event \\
\hline
events' = \langle ev? \rangle \frown events \\
\hline
\end{array}
$$

[3] This condition could be omitted since we declare *local_id* as a natural number but it is sometimes worthy to explicit such implicit conditions.

In the *Receive* schema, we just pop an event ($ev!$) from a given and non-empty input channel without forgetting to remove it effectively. The *Send* schema takes as input an event ($ev?$) and pushes it in front of the output channel event queue[4].

When received effectively, the events must be queued for later (asynchronous) treatment using the *InQueue* facility. Concurrently, events can be fetched at any (non-deterministic) time by the *InFetch* operation. The corresponding schemas are as follows:

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{InQueue}} \\ \Delta \mathit{Behavior} \\ ev? : \mathit{Event} \\ \hline \mathit{in\_queue}' = \langle ev? \rangle \frown \mathit{in\_queue} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{InFetch}} \\ \Delta \mathit{Behavior} \\ ev! : \mathit{Event} \\ \hline \mathit{in\_queue} \neq \langle\rangle \\ ev! = \mathit{last}\ \mathit{in\_queue} \\ \mathit{in\_queue}' = \mathit{front}\ \mathit{in\_queue} \\ \mathit{local\_id}' = \mathit{local\_id} + 1 \\ \hline \end{array}$$

Queuing consists simply in pushing the new event at the beginning of the queue (here, *in_queue*). Since we are, in the case of the *InQueue* schema, working at the *Behavior* level, the *Receive* schema shown previously must of course be promoted since communication channels are subparts of the considered state space. However, the promotion in this case does not impact on the concurrency issues we are discussing in this paper.The fetching phase can only be performed when the queue is not empty. The corresponding *InFetch* operation simply outputs the last event in the queue before removing it. From the previous schemas we can easily describe the output queuing system:

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{OutQueue}} \\ \Delta \mathit{Behavior} \\ ev? : \mathit{Event} \\ \hline \mathit{out\_queue}' = \langle ev? \rangle \frown \mathit{out\_queue} \\ \mathit{local\_id}' = \mathit{local\_id} - 1 \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{OutFetch}} \\ \Delta \mathit{Behavior} \\ ev! : \mathit{Event} \\ \hline \mathit{out\_queue} \neq \langle\rangle \\ ev! = \mathit{last}\ \mathit{out\_queue} \\ \mathit{out\_queue}' = \mathit{front}\ \mathit{out\_queue} \\ \hline \end{array}$$

The other operation involved in the process, *Exec*, has a more complex definition, mostly unrelated to our discussion. We will mainly retain that it serves as an interface between the input and output queuing systems, as depicted on Fig. 1. This execution phase denotes a finite sequence of four different kind of internal actions: functional action, side-effect, explicit as well as implicit event emissions. We are especially interested in the two last categories of actions concerning event communications. We will take into account that whenever an event is received and handled, then it denotes one or many resulting event emissions. This is captured by the signature $\mathit{Exec} \mathrel{\widehat=} [\Delta \mathit{Behavior};\ evs! : \mathrm{seq}\, \mathit{Event}]$ where all the events in $evs!$ must be "passed" to *OutQueue* for effective emission.

[4] These simplified schemas denote unfair FIFO channels whereas our model also support alternative transport semantics with fairness restrictions.

2.3 Proof obligations

In traditional Z-based specification methodologies, designers must conduct a set of formal proofs to verify incrementally the consistency of the system being modeled. In state/transition approaches like ours (or almost any Z-based model of our knowledge), this mostly consists in (1) *initialization theorems* to ensure that initializations preserve state invariants and (2) *precondition calculations* to enforce the consistency of the operations modifying the state space. We are mainly interested here in the potentially concurrent activation of operation schemas so that we will only address the issues concerning precondition calculations. These calculations must be done for each *Delta*-operation[5] defined on a given state. Establishing the list of all preconditions ensures that either the state invariant is completely preserved by the operation "effects" or that some other condition must be fulfilled. Depending on the context, this may lead to incrementally change the specification (by modifying the state definition) or just inform the system designers that the specification is somewhat incomplete (which occurs most of the time in complex systems where all the details cannot be taken into account).

For example, we can calculate the preconditions of *InQueue* as follows:

$$
\begin{aligned}
&\mathrm{pre}\ \mathit{InQueue} \Leftrightarrow \exists\, \mathit{Behavior}' \bullet \mathit{InQueue} && \text{[def. of pre]}\\
&\quad \Leftrightarrow \exists\, \mathit{in\_queue}, \mathit{in\_queue}' : \mathrm{seq}\, \mathit{Event} \bullet\\
&\qquad \exists\, \mathit{ev?} : \mathit{Event} \bullet\\
&\qquad\quad \mathit{in\_queue}' = \langle \mathit{ev?} \rangle \frown \mathit{in\_queue} && \text{[def. of } \mathit{InQueue}\text{]}
\end{aligned}
$$

We use here the regular definition expansion technique to rewrite the proof obligation, using the concerned state and operation schemas. Then, we simplify our calculation by putting all conditions at the same level (i.e. removing the local existential quantifiers) and replacing the dotted variables by their values (unique point rule) to obtain the following condition:

$$\langle \mathit{ev?} \rangle \frown \mathit{in\_queue} \in \mathrm{seq}\, \mathit{Event}$$

This is a theorem since *in_queue* is in essence a sequence of events. Consequently, there is no implicit precondition attached to the operation. The next step would be to explain what happen when the preconditions are not matched, in order to complete the operation.

3 Concurrent control model

As emphasized in [11] (Chap. 5), the Z notation provides great freedom regarding the operational interpretation of schemas. In consequence, any kind of system

[5] *Delta*-operation is our terminology for denoting an operation that modifies some system's state.

can virtually be conceived using Z-schemas. However, as time went by, the notation has been tightly connected to a well-tried methodology for the design and refinement of sequential systems.

In the previous section, we intuitively "imagine" from the state and operation schemas an operational model of our queuing system for components. The problem is that these schemas and the associated proof obligations only ensure a "correct" operationalization of the system as a sequential abstract machine. However, we are mainly concerned in our middleware research by the concurrent and distributed operationalization of such systems. Consequently, we propose in this section to somewhat change the intuitive interpretation of Z-based specifications by completing the functional requirements with a concurrent control model. This control model allows the description of *concurrent activations* of Z operations using *concurrent activation networks* represented by Petri-nets.

3.1 Petri-nets

Using the Petri-net terminology, we find within such activation networks : *places*, *markings* (using discrete *tokens*), *transitions* and *arcs* to link the places to the transitions (and vice-versa). More formally, a Petri-net N is defined as a quadruplet $N = (P,\ T,\ Pre,\ Post)$ where:

- P is the set of places,
- T is the set of transitions,
- $Pre : P \times T \rightarrow \mathbb{N}$ is the function of preconditions,
- $Post : P \times T \rightarrow \mathbb{N}$ is the function of postconditions,

The *marking state* of a Petri-net is described by a vector M of naturals counting the number of tokens inside places (the vector length is hence equivalent to the number of places in the net). We distinguish the *initial* marking at time 0 since the Petri-nets will evolve discretely over time with tokens going on and through transitions from places to places. A *place marking* for place p is the integer valued function, noted $m(p)$, counting the number of tokens at place p over time.

Fig. 2 shows two simple nets with similar properties, each one of them consisting in 3 places and 4 transitions. We will now describe the top network (input processing) in more formal terms. The places and transitions are defined as follows :

- $P = \{InQueue, InLock, InFetch\}$
- $T = \{ql, qr, fl, fr\}$

The initial marking state of this net is $M_{init} = [\ 0\ 1\ 0\ ]$ since only the *InLock* place is marked at time 0 (i.e. $m(InLock) = 1$).

The *Pre* function explains, for each transition and each place, how many tokens must be consumed so that it can be crossed. We usually represent it as a matrix. The *Post* vector explain complementarily the number of tokens produced when a transition is crossed. In our case, they are defined as:

$$Pre = \begin{bmatrix} 1\,0\,0\,0 \\ 0\,1\,1\,0 \\ 0\,0\,0\,1 \end{bmatrix} \quad Post = \begin{bmatrix} 0\,1\,0\,0 \\ 1\,0\,0\,1 \\ 0\,0\,1\,0 \end{bmatrix}$$

In these matrices, transitions are aligned in columns and places in rows. Suppose for example that we want to cross transition *fl* (third column) from the initial marking. In order to find the next marking state M', we calculate:

$$M' = M_{init} - Pre_3 + Post_3 = [\,0\;1\;0\,] - [\,0\;1\;0\,] + [\,0\;0\;1\,] = [\,0\;0\;1\,]$$

Hence, after crossing transition *fl*, *InLock* "gave" its token to *InFetch*[6]. The most seducing aspect of such Petri-nets, in addition to their simplistic denotation, concerns their implicit non-deterministic behavior. We only define here a set of preconditions and postconditions for transitions, without telling which of them will or will not get crossed effectively at a certain time. This is where the non-deterministic nature of Petri-nets appear.

3.2 Activation networks

Concurrent activation nets are introduced to design the control model of systems. They are refinements of traditional Petri-nets. We distinguish in such networks (1) the *utility* places which are standard Petri-net places and (2) the *ZPlaces* representing potential activations of operations in the Z-part of the specification. In the top network of Fig. 2, *InQueue* and *InFetch* places are *ZPlaces* whereas *InQueueLock* is a utility place. It has no direct impact on the functional specification but involves control-level changes that will have repercussions on the proof obligations.

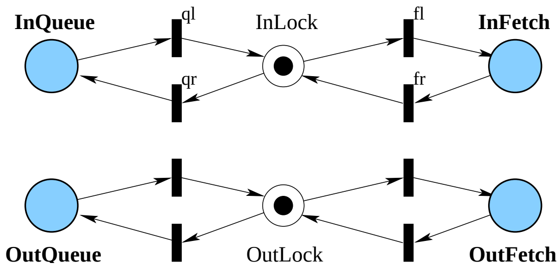

Fig. 2. Queuing phase concurrent activation networks

The basic rule of our concurrent activation network model is that if a given ZPlace contains a token, then, in the current state of the system, the corresponding operation (described by a Z schema) becomes *enabled*: its preconditions must

[6] Regarding the denotation, giving a token corresponds in fact to consuming a token and producing a new token.

be verifiable and the state changes it denotes (through its postconditions) must be taken into account.

For example, on the top network of Fig. 2, if the *fl* transition is crossed, then the *InFetch* operation becomes enabled so that, in Z terms, *pre InFetch* must be verifiable. We can also see that *pre InQueue* must be proven since this operation is also enabled (through transition *qr*) in the network. As a matter of fact, we can formally prove (either by simple flow analysis or, automatically, using one of the several available tools for Petri-nets analysis) on our example that the network is 1-bounded (there is at most one token in the system or $|| M || \leq 1$) so that only one place can be marked at a certain time.

In consequence, there is no concurrency involved and no new proof obligation emerges from the given model. The case is similar for the bottom network of Fig. 2 : we can prove formally that operations *OutQueue* and *OutFetch* are *mutually exclusive*, thanks to the use of a locking mechanism (the *OutLock* places and transitions).

3.3 Concurrency issues

The concurrent model we began to design is composed, for the moment, out of the two networks depicted on Fig. 2. Thus, these are not separated networks, they both are part of the *complete* concurrent control model of our system. Therefore, it is obvious in the resulting network that these two independent subnets are *parallel* in essence and, as so, denote concurrent activities. For example, it is likely that *InQueue* and *OutQueue* may become simultaneously enabled once the system is effectively operationalized. The same situation arises with operations *InFetch* and *OutFetch*, as well as *InQueue* and *OutFetch* and finally *InFetch* and *OutQueue*. The most important property of such concurrency constraints is that they can be automatically found using Petri-net analysis tools; the list of added proof obligations do not have to be established by hand[7].

All these concurrent activations must of course be taken into account at the functional level, under the form of new obligation proofs. For example, for operations *InQueue* and *OutQueue* to become enabled concurrently (and then noted $InQueue \; || \; OutQueue$), we must first calculate their independent preconditions *pre InQueue* and *pre OutQueue* but also their dependent ones which, in this case, correspond to $pre\,(InQueue \,\land\, OutQueue)$. The problem with the latter condition is that it does not capture well the postcondition effects when shared variables are involved. In that case, any order sequential composition must be established so that the generic form of the needed precondition calculation is the following one :

$$\text{pre } Op_1 \; || \; Op_2 \mathrel{\widehat=} \text{pre } Op_1 \land \text{pre } Op_2 \land \text{pre } (Op_1 \mathbin{;} Op_2) \land \text{pre } (Op_2 \mathbin{;} Op_1)$$

In our case, we simply have to calculate $pre\, InQueue \,\land\, pre\, OutQueue \,\land\, pre\,(InQueue \,\land\, OutQueue)$ since there is no postcondition sharing involved.

[7] Automated Petri-net analysis becomes complex and thus CPU-intensive when *model checking* is involved. As we can see in this paper, (fast) structural analysis is generally sufficent to deal with concurrent activation networks.

Moreover, we can note that both operations are also independent in term of preconditions. Hence, in that particular case : $pre\,(InQueue \wedge OutQueue) = pre\,InQueue \wedge pre\,OutQueue$. In consequence, there is no added proof obligation since we have already proven this in the functional specifications and its implicit sequential control model.

The same might be concluded about almost all the pairs of operations the two networks reference but this is not the case for the *InFetch* and *OutQueue* operations which seem to be coupled (in term of control) because of the shared *local_id* variable. The calculation of $pre\,InFetch \fatsemi OutQueue$, when removing the details except the shared conditions leads to the following statements:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{pre}\ InFetch \fatsemi OutQueue \Rightarrow \exists\, local\_id, local\_id', local\_id'' : \mathbb{N} \bullet} \\
\quad local\_id' = local\_id + 1 \wedge & [\text{from } InFetch] \\
\quad local\_id' \geq 1 \wedge local\_id'' = local\_id' - 1 \wedge & [\text{from } OutQueue] \\
\quad local\_id \geq 0 \wedge local\_id' \geq 0 \wedge local\_id'' \geq 0 & [Behavior \text{ invariant}]
\end{array}
$$

In order to prove the consistency of these conditions, we simply use the fact that $(local\_id \geq 0 \wedge local\_id' = local\_id + 1) \Rightarrow local\_id' \geq 1$.

However, the converse can not be proved that simply :

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{pre}\ OutQueue \fatsemi InFetch \Rightarrow \exists\, local\_id, local\_id', local\_id'' : \mathbb{N} \bullet} \\
\quad local\_id \geq 1 \wedge local\_id' = local\_id - 1 \wedge & [\text{from } OutQueue] \\
\quad local\_id'' = local\_id' + 1 \wedge & [\text{from } InFetch] \\
\quad local\_id \geq 0 \wedge local\_id' \geq 0 \wedge local\_id'' \geq 0 & [Behavior \text{ invariant}]
\end{array}
$$

The problem is easy to figure out : if *local_id* equals to 0, then the preconditions for *OutQueue* can not be verified. This introduces a control-level dependency between the two operations, *InFetch* having to come always before *OutQueue* when the value of the local identifier is zero. In other terms, the concurrent activation of the two functionalities can not be verified *in general*. To solve this problem, we both should try to work at the control and functional levels of our specification. At the functional level, we can use the precondition $local\_id \geq 1$ of operation *OutQueue* as a "marker" for the sequential dependency. At the control level, this would involve the connection, in some way or another, between the two activation networks, whose realization is shown in the next section.

4 Modularity and refinement

The key aspect of our proposal is that, added to its relatively small impact on the standard Z-based methodology, it has been mainly designed to support (and, in some way, enforce) the most important methodological tools conveyed by the Z schema-based notation : *modularity* and *refinement*.

4.1 Modular specifications

In the preceding sections, we described the activation network of the two concurrent queuing/fetching systems underlying the Comet middleware communication subtract. We haven't shown yet the reception process whose concurrent model is presented on Fig. 3.

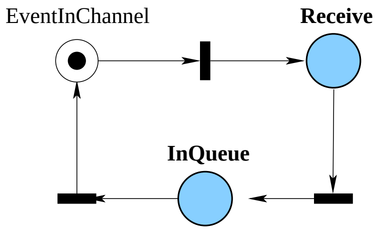

Fig. 3. Receiving phase concurrent activation network

It seems clear in this network that the *Receive* and *InQueue* operations are exclusive, the former having to be active at least once before the latter. This fact can be formally analyzed, automatically proving that for every marking m of this network, we have:

- mutual exclusion : $m(\mathit{Receive}) + m(\mathit{InQueue}) \leq 1$

Thus, we do not raise any new proof obligation using our formerly explained "discovery" method. It is now possible, from the networks of Fig. 2 and Fig. 3, with some extra information added, to deduce a global activation network, as shown on Fig. 4.

Of course, the resulting network adds some more transitions and places and is not just the concurrent composition of the modularly-defined subnetworks. The impact on the preconditions and postconditions functions is in that case purely *additive* : only *new* proof obligations may result from such assembly. Moreover, sequential proofs are particular cases (and sub-proofs) of concurrent ones so that all the local proofs will be *at least* used in the global network.

Table 1 summarizes all the proof obligations we finally raised using our method on the network shown on Fig. 4. Network invariants are obtained by classical flow analysis and lead to a set of inequalities. Two scenarios are also possible :

1. Atomic activations: there is only one possible mark for a set of places (first part of our table), so only one place in this set could be activated, which means that only one operation may be activated at a given time. In that case, all we have to do is to calculate each operation preconditions separately and merge them ($\wedge$ operator).

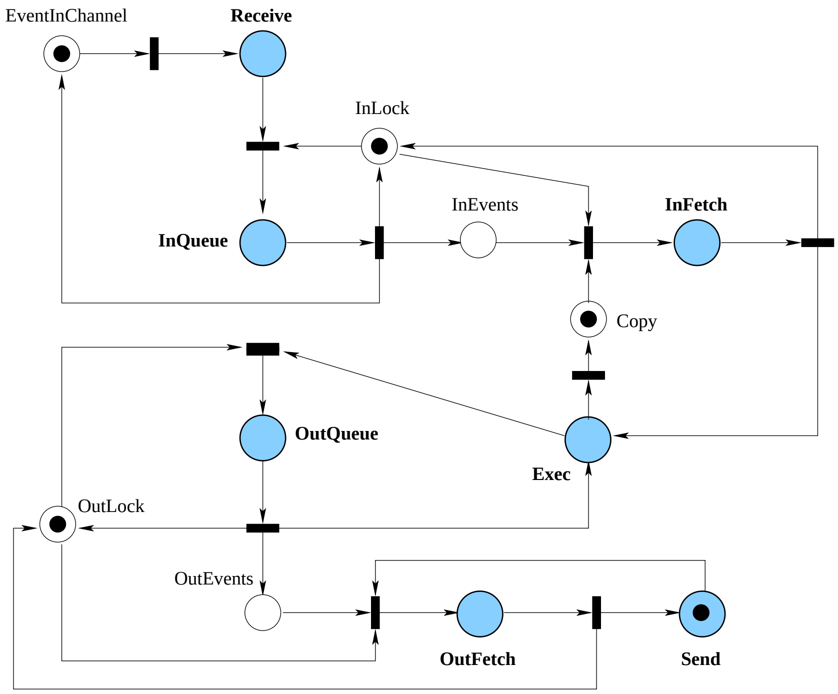

Fig. 4. Complete concurrent activation

2. Concurrent activations: there are multiple marks for a set of places (rest of the table), so several places could be activated, which means that several operations may become enabled simultaneously. In that case, we can't separate each operation preconditions evaluation.

At the functional level, the only impact of the activation networks concerns the new proof obligations. If a precondition calculus leaves holes in the specification, then this can lead to the modification of some functional aspect. However, except in the case of particularly inadequate specifications (regarding concurrency), only local state or operation schema modifications will be required to meet the concurrency model (e.g. synchronizations variables, timestamps, etc.). It is our belief that the general structure of the specification will be, most of the time, preserved even if the functional and control-level requirements are addressed separately (which should not be, in our opinion, the standard process).

Another interesting result we can draw out of the analysis of the whole activation network is that if there is only one event in the system, then the *InFetch* and *OutQueue* cannot be activated simultaneously. In simple terms, it is obvious that in order to have one token in the *OutQueue* ZPlace, then it has to be the "one" which "passed" through *InFetch* and *Exec* before. More formally, we can prove (automatically) that:

Network invariant	**Proof obligation**
Atomic activations	
$m(\mathit{Receive}) + m(\mathit{InQueue}) \leq 1$	pre $\mathit{Receive} \land$ pre $\mathit{InQueue}$
$m(\mathit{InQueue}) + m(\mathit{InFetch}) \leq 1$	pre $\mathit{InQueue} \land$ pre $\mathit{InFetch}$
$m(\mathit{InFetch}) + m(\mathit{Exec}) \leq 1$	pre $\mathit{InFetch} \land$ pre Exec
$m(\mathit{OutQueue}) + m(\mathit{OutFetch}) \leq 1$	pre $\mathit{OutFetch}$
$m(\mathit{OutFetch}) + m(\mathit{Send}) \leq 1$	pre Send
Concurrent activations	
$m(\mathit{Receive}) + m(\mathit{InFetch}) \leq 2$	pre $(\mathit{Receive} \mid\mid \mathit{InFetch})$
$m(\mathit{Receive}) + m(\mathit{Exec}) \leq 2$	pre $(\mathit{Receive} \mid\mid \mathit{Exec})$
$m(\mathit{Receive}) + m(\mathit{OutQueue}) \leq 2$	pre $(\mathit{Receive} \mid\mid \mathit{OutQueue})$
$m(\mathit{Receive}) + m(\mathit{Send}) \leq 2$	pre $(\mathit{Receive} \mid\mid \mathit{Send})$
$m(\mathit{InQueue}) + m(\mathit{Exec}) \leq 2$	pre $(\mathit{InQueue} \mid\mid \mathit{Exec})$
$m(\mathit{InQueue}) + m(\mathit{OutQueue}) \leq 2$	pre $(\mathit{InQueue} \mid\mid \mathit{OutQueue})$
$m(\mathit{InQueue}) + m(\mathit{Send}) \leq 2$	pre $(\mathit{InQueue} \mid\mid \mathit{Send})$
$m(\mathit{InFetch}) + m(\mathit{OutFetch}) \leq 2$	pre $(\mathit{InFetch} \mid\mid \mathit{OutFetch})$
$m(\mathit{InFetch}) + m(\mathit{OutQueue}) \leq 2$	pre $(\mathit{InFetch} \mid\mid \mathit{OutQueue})$
$m(\mathit{InFetch}) + m(\mathit{Send}) \leq 2$	pre $(\mathit{InFetch} \mid\mid \mathit{Send})$
$m(\mathit{Exec}) + m(\mathit{Send}) \leq 2$	pre $(\mathit{InFetch} \mid\mid \mathit{Send})$
$m(\mathit{OutQueue}) + m(\mathit{Send}) \leq 2$	pre $(\mathit{OutQueue} \mid\mid \mathit{Send})$

Table 1. Concurrent activation proofs

$$\mid\mid M \mid\mid \leq 4 \Rightarrow m(\mathit{InFetch}) + m(\mathit{OutQueue}) \leq 1$$

This invariant shows that the "unexpected" precondition we calculated at the end of Sect. 3.3, $\mathit{local\_id} \geq 1$, is "satisfied" by the above invariant if there is only one event in the system (which was the conflicting issue).

4.2 Refinement

The Z notation generally denotes a top-down specification process from abstract models of requirements to the resulting implementation of the system. The mediation of a *refinement calculus* allows this incremental implementation of the specified system. Refinement, in the case of Z, can be used to produce more concrete states and/or operations from more abstract ones. [11] is in our opinion a good source of information for the use of refinement in the framework of the Z notation and tools.

An important aspect of state or operation refinements is that they generally concern a reduced set of well localized schemas so that, once again, the modularity of the specification at the more abstract level is not put into question by the refinement needs. In our case, the main consequence is that the added proof obligations regarding a given activation network remain the same but must be redone in two situations:

1. The proof obligation references an operation that has a new refined version, or
2. the state referenced by the operation has been itself refined.

Unsurprisingly, this has some impact on the proof part of the specification. First, the refinement proofs must be conducted, which is in its own a complex process. Thereafter, concurrently activable operations must be checked against the refined state. We do think, however, that the overall structure of the specification remains unchanged in the refinement case. The operations preserve, in a way, their "locality" in the model so that the rules for concurrent activability do still apply in the refined case. Without an explicit denotation for the interface between the functional and control models, it is however necessary to consider the refined cases as "fresh" ones.

On the concurrent side, it is of course possible to refine an activation network to match the refined functional requirements. In the case of our middleware system, we propose various refinements. We will describe in this section an alternative *multi-session* behavior model for components. A multi-session controlled component allows internal concurrency, which means in our system the parallel handling of events. At the functional level, this leads to both the refinement of the component *Behavior* schema, as well as of most of the operations involved in the process of input and output events. For example, the following schema denotes the refinement of the main state schema:

$$
\begin{array}{|l}
\hline
\textit{MSessionBehavior} \\
\hline
\textit{Behavior} \\
\textit{ssessions} : (\textit{TYPE} \times \textit{IDENTIFIER}) \mapsto \textit{MSession} \\
\textit{gsessions} : \textit{TYPE} \mapsto \textit{MSession} \\
\hline
\#(\mathrm{ran}\ \textit{ssessions}) + \#(\mathrm{ran}\ \textit{gsessions}) \leq \textit{MSession\_max} \\
\forall\, s : \mathrm{dom}\ \textit{ssessions} \bullet \exists\, \textit{in} : \textit{in\_channels} \bullet \\
\quad \exists\, \textit{ct} : \textit{in.type} \bullet \{\textit{first}(s) \mapsto \textit{ct}\} \subseteq \textit{subtype} \land \textit{second}(s) = \textit{in.id} \\
\hline
\end{array}
$$

This new state schema refines the more abstract *Behavior* state by adding concurrent sessions, either *specific* or *generic*, in the event management system. Specific sessions distinguish event sources and types whether generic ones only use event types (and are thus shared by all sources). Without going in too detailed explanations, the intention here is that if an event of the given type (or a subtype[8]) is received and if the source matches (specific sessions only), then it will be processed concurrently. There exists several sub-state schemas to define this whole state refinement. We will only give the *MSession* state which describes a local session:

[8] The subtyping relation we use is a relation *subtype* whose signature is $\mathbb{P}\, \textit{TYPE} \times \textit{TYPE}$.

$$SESSION\_TYPE ::= specific \mid generic \mid none$$

$$\begin{array}{|l} EmptySession : MSession \\ \hline EmptySession.stype = none \end{array}$$

$$\begin{array}{|l} MSession\_max : \mathbb{N} \\ \hline MSession\_max = 10 \end{array}$$

$$\begin{array}{|l} \textbf{MSession} \\ stype : SESSION\_TYPE \\ in\_queue : \mathrm{seq}\, Event \\ out\_queue : \mathrm{seq}\, Event \end{array}$$

These schemas show that sessions are composed out of a categorical type (either *specific*, *generic* or *none*) as well as an independent pair of queuing systems. Using this refinement (easily promoted to the independant global behavior), we can provide session-specific versions of the event management facilities such as the input queuing phase:

$$\begin{array}{|l} \textbf{MSessionInQueueSession} \\ \Delta MSession \\ event? : Event \\ session? : MSession \\ \hline session? \neq EmptySession \\ in\_queue' = \langle event? \rangle \frown in\_queue \\ out\_queue' = out\_queue \end{array}$$

$$\begin{array}{|l} \textbf{MSessionInQueueDefault} \\ Receive \\ \Xi MSession \\ session? : MSession \\ \hline session? = EmptySession \end{array}$$

The left schema describes the case when an event is matched positively with a session so that it is handled specifically (i.e. in parallel). When it is not matched, then the *MSessionInQueueDefault* operation tells the system the event has to be processed at a higher state level than the session one. There are of course several new proof obligations raised by this refinement of the more abstract specification. But we know from the previous sections how to find them.

There is however one aspect we haven't yet given a real meaning: the concurrent handling of event in sessions. We only shown yet functional support for concurrency but the concurrent model itself lacks concurrency. Our proposal here is to refine this concurrent model, giving a refinement of the more abstract activation network of Fig. 4. For clarity, we only show the result in term of input queuing, fetching and dispatch on Fig. 5.

We introduce here a way to link the functional and concurrent models more tightly using natural number globals. The *MSession_max* variable (which has of course to be a natural) is related to the homonymous place (called an *NPlace* in our terminology) of the activation network. The trivial network invariant $m(MSessionInQueueSession) \leq m(MSession\_max)$ induces that the *MSessionInQueueSession* operation in the functional specification can be simultaneously activated N times (if the marking is exactly N). That is, multiple instances of the same operations may be active at a certain time. It is of course

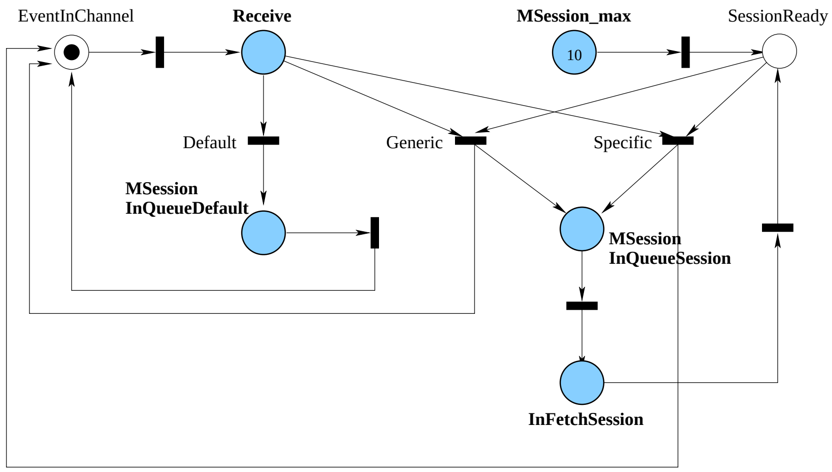

Fig. 5. Refined activation network (input only)

insufficient to prove that one activated instance fulfills the invariant requirements. We denote each different activation of the same operation by an integer subscript so that we have to check the following concurrent activation:

$$InQueueSession_1 \;||\; InQueueSession_2 \;||\; ... \;||\; InQueueSession_N$$

From the local point of view, each instance of the operation takes is own state as the *session*? variable. They are thus completely independent with each other for any given session (i.e. the scope of $\Delta MSession$ is local) so that finally, no new proof obligation is needed in this particular case.

Let us now consider the *InFetchSession* operation and an extract of its promotion to the global state:

InFetchSession

$\Delta MSession$
$ev! : Event$

$in\_queue \neq \langle\rangle$
$ev! = last\ in\_queue$
$in\_queue' = front\ in\_queue$

InFetchSessionPromote

$\Delta Behavior$

$local\_id' = local\_id + 1$

Here, we see that the promoted operation has an impact on the local identifier. So, it is not so simple to prove the consistency of:

$$InFetchSession_1 \;||\; InFetchSession_2 \;||\; ... \;||\; InFetchSession_N$$

However, all sequential interleaving of these concurrent activations only concern the shared *local_id* variable and thus denote the following condition:

$$\exists\, local\_id : \mathbb{N} \bullet$$
$$(\exists\, local\_id_1 : \mathbb{N} \bullet local\_id_1 = local\_id + 1 \wedge local\_id_1 \geq 1) \wedge$$
$$\ldots \wedge$$
$$(\exists\, local\_id_N : \mathbb{N} \bullet local\_id_N = local\_id_{N-1} + 1 \wedge local\_id_N \geq N)$$

It is not difficult to see that this is a theorem, but this clearly shows that the concurrency point of view has, as expected, some repercussions on the process to establish the consistency of the system, if compared to the standard sequential point of view.

5 Related work

To our knowledge, while there exist specifications of concurrent programs in Z, there were only few attempts to address concurrency issues from a methodological point of view. Most of the time, concurrency proofs must be established completely by hand which is a tedious practice. Predicative notation has been integrated with Z in [7] to allow the design of concurrent and communicating protocols as well as deriving proof methods for concurrent aspects. Despite the interesting results of this approach, its use of a somewhat new and not very spread extension language may explain its relatively reduced impact on the Z community. Combinations of process algebras with Z [3] or Object-Z , as in TCOZ [6], seem to have received more audience. These approaches integrates fine-grain parallelism constructs (as found in CSP and CCS) with state-based functional specifications. Of course, the impact of the extra-features for concurrency and communication on the full-featured Z notation (or object-oriented extensions) is important. New and somewhat complex design and proof techniques must be introduced. This also reveals that concurrency is in essence a cross-cuting concern in presence of complex functional requirements. We diverge here since our concurrent constructs (activation networks) do not belong to the functional specification; they are kept separate. Therefore, the concurrency issues are designed (and partially verified) independently, which is not the case with integration techniques. Moreover, we do not introduce communication since we focus on shared-memory models. We then think process algebras may be used complementarily for aspects dealing with distribution.

The first attempt to bring closer the Z and Petri-net languages seems to be [5]. In order to preserve the advantages of both formal methods in their respective domains (in our opinion complex functional requirements on one side and complex concurrent models on the other side), the authors propose to maintain a synchronized model in both languages. The synchronizations points are transitions of the Petri-net which are associated to operation schemas so that the state of both specifications can be synchronized (the transitions are the same). Some proof techniques can be derived from this model but it seems difficult to maintain a clean separation at that level. Moreover, the Petri-nets can become very large because they have to reflect properly the functional requirements expressed in the Z part of the model. In our approach, we wanted to limit as much as possible the size of the networks (so that automated proof could be conducted as often as possible) and also limit the impact of the concurrent aspects on the specification. Thus, we only model activation rules in the network —as synchronized places— whereas functional requirements are only expressed in the Z part of the specification. The impact is then restricted to the proof level because concurrent activations only denote new proof obligations.

6 Conclusion

We have presented in this study a method we employed in a *real*, middle-sized software development process to support the traditional Z formal method when *concurrency* is involved. The first contributing aspect of our approach is its focus on having a minimal impact on the notation and method itself. We do not extend the schema-based notation and only add one operator for precondition calculations of concurrently enabled operations. In consequence, the full power of Z can be used to model the functional requirements of the systems, namely its state and operations. Modularity and support for refinement are not compromised and can be freely used.

However, concurrent systems are *in essence* different (and somewhat more complex) from sequential ones. Hence, an expressive and efficient (i.e. easily analyzable) language to model concurrency is needed. We think that Z-extensions for concurrency, despite their interest, are more complex and more intrusive than ours. Our contribution, here, is not to propose a new language (or extension) but simply to *reuse* an existing and widely spread language, *specifically aimed* at modeling concurrent systems, namely Petri-nets. We only show how to interpret such networks in the framework of the Z method. This allows system designers to inherit a large palette of tools for automated invariant calculations as well as work from a large and well established community.

The clean separation between functional and control-level concerns, as proposed by this hybrid formal method, also ensures that the concurrent activation networks remain "manageable". By removing all the purely functional aspects of the specifications, we limit in great proportions the state-space of the Petri-nets so that the usual analysis tools can be used even in the case of fairly complex specifications. In our case study, we only manipulated low-level Petri-nets with arcs of weight 1 and sizes of never more than dozens of places and transitions. Of course, our confidence in the resulting concurrent system must be tempered by the fact we conducted all Z proofs by hand.

This points out the main drawback of our approach : we support the automated analysis of the concurrent side of the model only. Some automation can be obtained at the functional-level of the specification, using existing Z-based tools. But at the interface between the two models, synchronization is done by hand. Moreover, the denotation of this interface is not easy to figure out, given the different essence of set-based theories (underlying Z proofs) and linear algebras (Petri-net analysis). Our approach here was to give a simple denotation of Petri-nets using the Z notation. But in order to obtain a complete and consistent denotation, we should be able to conduct all the proofs using the same framework, a task hardly manageable in our opinion. For example, it is arguable that despite the fact that our network are dead-lock and live-lock free, the Z schemas themselves can introduce functional-level dependencies that have an impact in term of concurrency. This, however, will be exposed when concurrent activation proofs will be conducted, resulting in new preconditions. And so forth, we surely cannot use automation tools at that level. But in practice, the pragmatic point of view we conveyed throughout the experiment led to an operational system, fulfilling our requirements; this showing that the method, if not mature, is valuable.

References

1. G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, 1986.
2. J.-P. Briot. *Actalk : a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment.* In S. Cook, editor, *Proceedings of ECOOP'89*, pages 109–129. Cambridge University Press, July 1989.
3. C. Fischer. How to combine z with a process algebra. In *Proceedings of ZUM'98*, volume LNCS 1493. Springer-Verlag, September 1998.
4. M. Heiner. Petri net based system analysis without state explosion. In *Proceedings of High Performance Computing'98*, April 1998.
5. M. Heiner and M. Heisel. Modeling safety-critical systems with z and petri-nets. In *Proceedings of Computer Safety, Reliability and Security*, volume LNCS 1698. Springer-Verlag, 1999.
6. B. Mahony and J. S. Dong. Timed communicating object z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.
7. R. F. Paige. Specification and refinement using a heterogeneous notation for concurrency and communication. Technical report, York University, October 1998.
8. F. Peschanski. A reflective middleware architecture for adaptive, component-based disitrubted systems. *IEEE DS Online*, 1(7), 2001.
9. F. Peschanski. A versatile event-based communication model for generic distributed interactions. In *Proceedings of DEBS'02 (ICDCS International Workshop on Distributed Event-based Systems)*. IEEE, July 2002.
10. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.
11. J. M. Spivey. *The Z Notation : a reference manual.* `http://spivey.oriel.ox.ac.uk/~mike/zrm/`, 2001.