IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 27, 2020, accepted June 17, 2020, date of publication June 22, 2020, date of current version July 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3004262

A Generalised Coordination Design Pattern for the EX-MAN Component Model

TAUSEEF RANA[1] AND ABDULLAH BAZ[2], (Senior Member, IEEE)
[1]Department of Computer Software Engineering, Military College of Signals (MCS), National University of Sciences and Technology, Islamabad 21955, Pakistan
[2]Department of Computer Engineering, College of Computer and Information Systems, Umm Al Qura University, Makkah 21955, Saudi Arabia

Corresponding author: Tauseef Rana (tauseefrana@gmail.com)

This work was supported by the Deanship of Scientific Research at Umm Al Qura University under Grant 19-COM-1-01-0015.

ABSTRACT In the latest technologies for next generation using in Cyber-Physical Systems, 5G and IoT (Internet of Things) based solutions have a significant contribution. For the construction of such applications, component-based development approaches offer to produce systems by using pre-built tested and reliable components with shorter development time. At the architecture level, a software system can be viewed as a collection of two kinds of elements. One kind is responsible for computation and the other kind is responsible for communication. Using a component model, that separates the communication and computation into distinct layers, enables us to secure the communication part of the system. In this paper, we propose a design pattern which defines coordination/communication program units (referred to as exogenous connectors) for a repository of reusable connectors in the EX-MAN component model. There are many attempts of implementing exogenous connector in different tools in unspecified ways. Our proposed pattern for a generalised exogenous connector helps in specifying exogenous connectors with enough details that can be used for the implementation of these connectors. Our model enables in-depth analysis of different kinds of exogenous connectors with respect to its static/dynamic behaviour in a system. In this paper, we model and simulate the static/dynamic behaviour of sample exogenous connectors based on our proposed model. Using our specifications of exogenous connector, we have developed exogenous composition framework (ECF) for system development.

INDEX TERMS Coordination, control flow, design pattern, communication, code generation.

I. INTRODUCTION

System development approaches are revised for quicker and safer construction with the change in technology and market trends. The technologies eminent for future growth include 5G and IoT based applications [5], [16]. Hence, software based systems are becoming bigger and complex with the aforementioned advancements of technologies. For quicker and economical development, the use of component based development (CBD) approaches for these technologies is also rising [4], [17]. The security and verifiability are two important features to achieve in CBD approaches [49]. Parry and Wolf proposed a model for software architecture in their seminal work [35]; this model is comprised of units representing computation (referred to as components) and communication (referred to as connectors). Two or more computation units are composed by using connectors to create a system. Hence, to address the issue of complexity and scalability, composition is used to create bigger and complex systems from existing program units [43].

The associate editor coordinating the review of this manuscript and approving it for publication was Ilsun You.

Software composition is the way to reduce the construction cost in CBD. Furthermore, CBD also seeks to automate composition as much as possible [26]. A component model defines a basic program unit (referred to as a component) and mechanisms to create bigger units/systems from smaller units (referred to as composition). Hence, for its focused *development for/with reuse* [27], CBD [18], [45] represents a paradigm to achieve reusability.

In CBD, many software component models are defined [29] for system development. With these models, reuse can be achieved with respect to computation and communication; in CBD many approaches are based on reusing pre-defined program units for computation (known as components) and program units for communication (known as connectors) [2], [30], [35], [48]. The work of researchers in [30] show the importance of the software connector domain.

Some component models (e.g. ACME [13], the X-MAN component model (X-MAN) [21], [25], [28]) define connectors as separate program units than components. X-MAN defines a separate layer of connectors for coordination/communication from the computation layer. This makes X-MAN a strong candidate for the latest technologies based systems with secure communication capabilities. Exogenous connectors proposed in X-MAN represent control coordination for the execution of computations offered by the components. Exogenous connectors can compose two or more components and adopt components in a system. There are a number of tools to provide support for exogenous connectors [23], [24], [48]; however, the connector semantics are defined in unspecified ways. Addressing the limitations of X-MAN and proposing a number of new extensions, an extended X-MAN (EX-MAN) is defined in [37], [38].

The emergence of design patterns [12] in software engineering has introduced a new way to achieve reuse in the software design phase. In software engineering, a design pattern is defined as a reusable solution for a reoccurring problem in a context. The obvious usefulness of a design pattern is the design knowledge with some confidence by the developers. In the context of CBD, many researchers have proposed new approaches by using patterns [6], [24], [44], [48].

In CBD, there are some component models in which connectors are defined as separate units than components [3], [13], [14], [36], [46]. In X-MAN, these connectors are referred to as exogenous connectors. However, these connectors are defined in a number of tools [23], [24], [48] in unspecified ways. Similarly, there are different kinds of connectors in the Reo component model [3]. Using a different version of X-MAN, some more connectors are defined in [44]. Despite many similarities in the connectors from the aforementioned approaches, these connectors are defined and implemented differently. This difference can cause some inconsistency in the way these are designed. Moreover, more efforts at the implementation level are needed. In this paper, we propose a generalised pattern which can ensure consistency in the design of these connectors. Coordination based composition concept is used in Reo, X-MAN, EX-MAN and used for web service composition. One benefit of our proposed pattern is the availability of a common framework for coordination based composition. With the help of this pattern, developers efforts in implementing connectors is eased. The proposed pattern is used to create all defined connectors in EX-MAN with enhanced features.

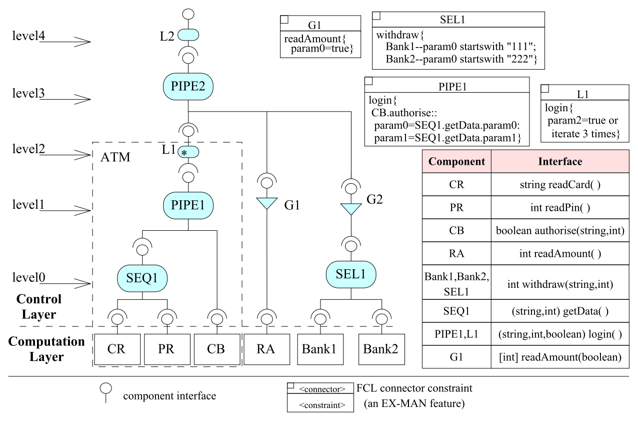
In this paper, for EX-MAN, we define a generic exogenous connector pattern which is used to specify exogenous connectors in enough details for their implementation. The purpose of this pattern is to reduce the complexity of control coordination at the generic architecture level. This is needed to analyze the behaviour of exogenous connectors at a level of abstraction. Furthermore, such a model can also be utilized as a base to construct a connector generator framework; this is out of scope for this paper. The generic pattern definition for connectors provides the behavioural semantics of exogenous connector in two phases: (i) the system construction (deployment) phase and (ii) the system execution (rum-time) phase. We use Coloured Petri net (CP-net or CPN) [19] to verify the behaviour of exogenous connectors in the deployment and the run-time phases. In future work, our work can be extended to generate as many connectors possible from [48].

The scope of this paper is to define a generic exogenous connector pattern for the EX-MAN component model. Section II introduces the EX-MAN component model with enough details for exogenous connectors. In Section III, after briefly introducing CP-net and CT-net, we define a generic exogenous connector's structure in terms of its properties, CP-net elements (places, transitions and arcs) and behaviour (set of functions) to extend the connector in the deployment phase. The behaviour (generation of component interface and the control/data flow to the connected component(s)) of exogenous connectors are defined and described (with examples) in Section IV. The sample CPN models of two exogenous connectors along with their simulation outcomes is shown in Section V. The defined exogenous connectors are implemented in a tool; this tool is described very briefly in Section VI. Section VII sheds light on the related work and Section VIII provides directions for the future works.

II. EXOGENOUS CONNECTORS IN EX-MAN

The EX-MAN component model (EX-MAN) [37] is based on X-MAN [25], [28]. The distinguishing feature of these models is the use of exogenous connectors to construct the communication part of a system. In X-MAN, exogenous connectors are defined in abstraction and the exact behaviours of these connectors are implemented in different tools [23], [24], [48]. EX-MAN extends and specifies exogenous connectors more rigorously. Primarily, there are two kinds of exogenous connectors: adaptors and composition connectors. In a system, the role of exogenous connectors can best be explained with the help of a working example. Hence, for this purpose, we consider a simple example of ATM system shown in Figure 1. In a system, EX-MAN exogenous connectors are annotated by constraints written in flow constraint language (FCL) [40]. To avoid complexity, the design is made simple; FCL constraints and service interfaces of all connectors can be found in [37].

In the shown system example, one ATM subsystem serves two different branches of a bank. The system design (based on two layers) is comprised of components and connectors connected in a hierarchy. The system has four composition connectors (sequencer SEQ1, selector SEL1, pipe PIPE1 and pipe PIPE2), four adaptor connectors (finite loop L1, infinite loop L2, guard G1 and guard G2) and five components (CR to read card number, PR to read pin code, CB to authenticate ATM card, RA to read withdraw amount, Bank1 and Bank2). Sequencer and pipe connectors passes the control (and data) to each connected component in sequence from left to right. A pipe is a special sequencer that can pass execution result of one component as input data for later component executions.



```
G1
readAmount{
  param0=true}

SEL1
withdraw{
  Bank1--param0 startswith "111";
  Bank2--param0 startswith "222"}

PIPE1
login{
 CB.authorise::
  param0=SEQ1.getData.param0:
  param1=SEQ1.getData.param1}

L1
login{
 param2=true or
  iterate 3 times}
```

Component	Interface
CR	string readCard( )
PR	int readPin( )
CB	boolean authorise(string,int)
RA	int readAmount( )
Bank1,Bank2, SEL1	int withdraw(string,int)
SEQ1	(string,int) getData( )
PIPE1,L1	(string,int,boolean) login( )
G1	[int] readAmount(boolean)

FIGURE 1. An ATM system in EX-MAN.

A selector passes control to only one component. A finite loop connector passes control to its connected component for fixed iterations and an infinite loop passes control to its connected component indefinitely. Based on a fixed criteria, a guard connector passes the control to its connected component.

For the system execution, in ATM PIPE1 reads an ATM card details (read by SEQ1 that passes control to CR and then to PR) and gets the authentication by passing card details to the central bank component CB. Finite loop connector L1 iterates 3 times to read ATM card and terminates iterating if the card is authenticated. After successful authentication, PIPE2 transfers control to G1 to read withdraw money amount from RA. After that PIPE2 passes control to G2 to select a bank by SEL1. G1 and G2 check the successful authentication. After serving one customer, loop connector L2 repeats the execution to serve the next customer. FCL constraints of four connectors are shown.

In connector PIPE1, an FCL constraint for 'login' service is defined. In this constraint, results of a service ('getData') from SEQ1 is passed as argument to a service ('authorise') of CB. The first iteration of connector L1 is unconditional; for every next iteration, the L1 constraint is checked. The L1 loop terminates if the output of 'login' service is 'true' or the loop runs for 3 times. Guard connector G1, passes control to the adapted component RA if the card was authenticated (represented by first argument param0). For the selector SEL1, the constraint is to pass control to the respective bank branch based on the value in the first argument of 'withdraw' service. Component Bank1 is selected if the value of the input parameter starts with string "111".

In CBD, the idealised component life cycle [22] (shown in Figure 2) shows the 3-phase life cycle of components. The first phase is referred to as component design phase; in this phase, by using a builder, basic components are programmed or composite components are created by composing existing components from the component repository. For system construction in EX-MAN, exogenous connectors plays a vital role in the deployment and run-time phases; hence, exogenous connectors have different semantics for

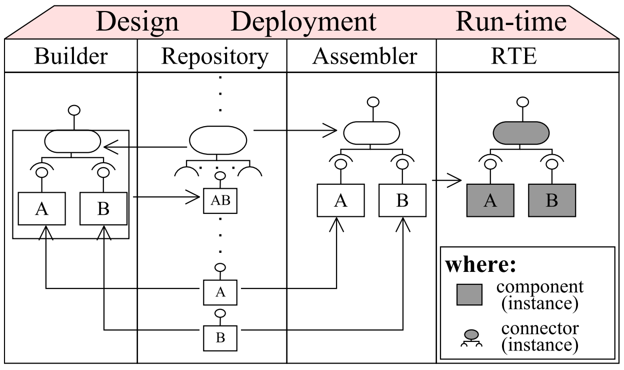


FIGURE 2. The idealised component life cycle.

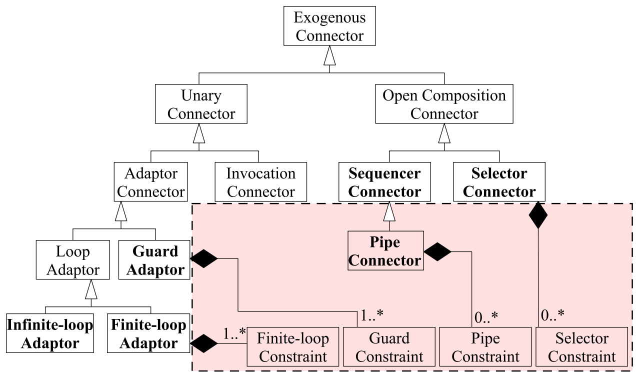


FIGURE 3. A conceptual model of exogenous connectors [40].

these two phases. For example, in the deployment phase by using assembler, exogenous connectors define the interface of composite (interface of SEQ1 composite) and adapted (interface of G1 adapted) components. In the run-time phase, exogenous connectors define the flow of control/data to the connected components.

In EX-MAN systems, for maintaining their existence in the execution phase, connectors and components are referred to as first-class elements of the model. The conceptual model for exogenous connectors is shown in Figure 3 by using the UML class diagram notation. The extension of EX-MAN are shown in the dashed rectangle and connectors shown in bold are used for system construction. Four of these can have FCL constraints.

A connector is a program unit that plays dual role in a system. In the design phase, based on the constraints, a connector defines the interface generation behaviour in the design phase. Similarly, based on the constraints, connectors define the control/data flows to the connected components and connectors in the run-time phase.

III. A GENERIC EXOGENOUS CONNECTOR

Operational semantics of workflow patterns in *newYAWL* [42] (a business process modelling language founded on workflow patterns) are defined by using Coloured Petri net (CP-net) [19] semantics. Exogenous connectors

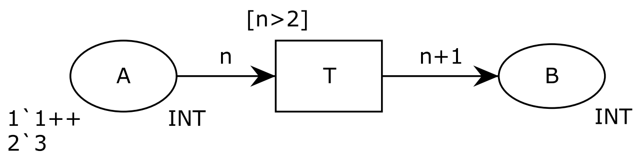


FIGURE 4. **An example in CPN tools.**

in run-time phase correspond to these workflow patterns. Hence, in order to define the semantics of exogenous connectors rigorously, we define a generic exogenous connector by extending Connector Template net (CT-net) from [24]; CT-net is a special kind of CP-net. For uniformity, we use CT-net to define both the deployment and the run-time (operational) semantics of exogenous connectors.

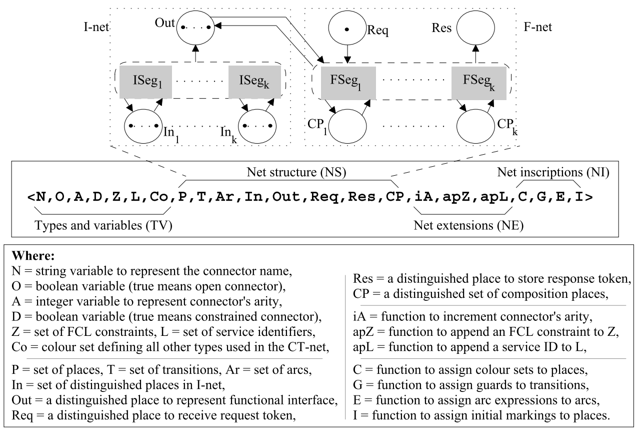
In this section, after briefly introducing CP-net and CT-net, we define a generic exogenous connector's structure in terms of its properties, CP-net elements (places, transitions and arcs) and behaviour (set of functions) to extend the connector in the deployment phase.

A. COLOURED PETRI NET

Coloured Petri net (CP-net) is a graphical-oriented language [20] which can be used to design, specify, simulate and verify different kind of systems [9]. Formally, at an abstract level, CP-net is a tuple (*NS, TV, NI*) of net structure (*NS*), types and variables (*TV*), and net inscriptions (*NI*). At the detail level, CP-net is a nine-tuple (P, T, A, $\sum$, V, C, G, E, I). Set of places (P), set of transitions (T) and set of arcs (A) represent net structure (*NS*). Set of colour sets ($\sum$) and set of variables (V) represent types and variables (*TV*) of the net respectively. Functions C, G, E and I represent net inscriptions (*NI*) where: (i) C assigns colour sets to places, (ii) G assigns guards to transitions, (iii) E assigns expressions to arcs, and (iv) I assigns initial markings to places. In this section, CP-net primitives are described briefly with the help of an example modelled in the CPN tools, as CP-net primitives are used to define CT-net.

In the example CP-net shown in Figure 4, there are two places (A and B with colour *INT*), one transition (T with a guard) and two arcs (arrowed lines with expressions) between the two places and one transition. The example CP-net in CPN tools before and after simulation is shown in Figure 30. Initial and current (shown in a box) markings of A are shown in Figure 30(a). Transition T is enabled as there are tokens in the input place (A) to fulfil *T's* guard. On firing the enabled transition, two tokens from place A are moved to place B, as shown in Figure 30(b).

In a net, a place represents a memory location to store tokens (data). Initial marking of a place (shown as a label next to the place Figure 30(a)) represents number of tokens and their values; this is required to simulate the net. In the net, tokens and their values shown in a box (next to a place) represent the current marking of the place. In the place marking, number of tokens and their value is separated by a backward single quote symbol; tokens with different values



<N,O,A,D,Z,L,Co,P,T,Ar,In,Out,Req,Res,CP,iA,apZ,apL,C,G,E,I>

Where:
N = string variable to represent the connector name,
O = boolean variable (true means open connector),
A = integer variable to represent connector's arity,
D = boolean variable (true means constrained connector),
Z = set of FCL constraints, L = set of service identifiers,
Co = colour set defining all other types used in the CT-net,

P = set of places, T = set of transitions, Ar = set of arcs,
In = set of distinguished places in I-net,
Out = a distinguished place to represent functional interface,
Req = a distinguished place to receive request token,

Res = a distinguished place to store response token,
CP = a distinguished set of composition places,

iA = function to increment connector's arity,
apZ = function to append an FCL constraint to Z,
apL = function to append a service ID to L,

C = function to assign colour sets to places,
G = function to assign guards to transitions,
E = function to assign arc expressions to arcs,
I = function to assign initial markings to places.

FIGURE 5. **CT-net.**

in a marking are separated by '++' symbol. The current marking of all the places in a net together represents the marking of the net, as shown in Figure 30(b). Transitions are enabled if there are tokens in the attached input places. When a transition is fired, tokens are moved from the input places to the output places. Transitions can have an optional guard expression.

In a net, arcs can represent flow of tokens and dependency between places and transitions. It is not permitted to have arcs between two places or between two transitions. Tokens can be modified by expressions on the arcs; for example, expression on arc between T and B in Figure 4 increments the token value.

B. CONNECTOR TEMPLATE NET

In EX-MAN, a composition connector in the deployment phase allows adding components to a composite. CP-net semantics and CPN tools do not support to extend a net dynamically (during simulation). Connector Template net (CT-net) [24], [32], [47] is a special kind of CP-net that defines the control flow for composition connectors. For system construction, to create current exogenous connectors with fixed arity (for unary connectors) or with open arity (for composition connectors), we extend CT-net by adding functions which can represent the computation to refresh/extend the net. Moreover, unlike the original definition of CT-net, we extend CT-net to create the component's interface. Furthermore, we define CT-net to create unary as well as composition connectors. Our CT-net is defined as a tuple (Figure 5).

Adopting a top-down approach, at an abstract level, CT-net can be defined as a tuple (*TV, NS, NE, NI*) of types and variables, net structure, net extensions, and net inscriptions. At the detail level, tuple element *TV* represents 7-tuple (*N, O, A, D, Z, L, Co*), element *NS* represents 8-tuple (*P, T, Ar, In, Out, Req, Res, CP*) or 2-tuple (*I-net, F-net*), element *NE* represents 3-tuple (*iA, apZ, apL*), and element *NI* represents 4-tuple (*C, G, E, I*). For simplicity, in our definition, we include necessary elements required to define the core behaviour of exogenous connectors. For example, we do not include elements to represent a connector's instance name and service annotations in the connector's interface in the definition of the generic exogenous connector. Similarly, functions to add service annotations, to rename a service and to evaluate/modify FCL constraints are also not included. In Figure 5, we show an example of a connector with arity 'k'; to avoid cluttering, we do not show place types, arc expressions, transition guards and place markings.

1) I-NET

I-net (a sub-net of CT-net) of a connector defines how a connector creates the interface of a component in the deployment phase. In *I-net*, cardinality of *In* is equal to the arity of the net ($|In| = A$). Each place in *In* is a source and destination for one arc; number of tokens in the initial marking of an *In* place is maintained at all times.

Place *Out* is a distinguished place of the net, such that *Out* is a destination for one arc and source for none. Tokens in the *Out* place represent selected services for propagation. Tokens in an *In* place represent tokens from the *Out* place of the root connector's *I-net* of a connected component.

In abstraction, in *I-net*, the dashed rounded rectangle can be viewed as a set of net segments. A segment consists of places, transitions and arcs; a segment is responsible for reading service tokens from an *In* place. In *I-net*, all segments are connected via some common net elements.

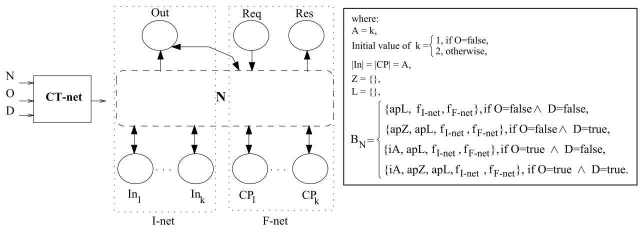
2) F-NET

F-net (a sub-net of CT-net) of a connector defines the control/data flow to the connected component(s) in the run-time phase. In *F-net*, *Req* is the distinguished place of the *F-net*, such that *Req* is not a destination for any arc but a source for exactly one arc in the *F-net*. Similarly, *Res* is a distinguished place of the *F-net*, such that *Res* is not a source for any arc in the *F-net* but a destination for one arc in the *F-net*. *Req* receives a service request token from outside *F-net* and *Res* receives a response token from inside the *F-net*. Element *CP* is a set of composition places with cardinality equal to the arity of the net ($|CP| = A$). A member of *CP* set ($CP_x \in CP$, where x is a subscript number) has exactly two distinguished places ($|CP_x| = 2$) for connection with *Req* and *Res* places of a connector.

In abstraction, in *F-net*, the dashed rounded rectangle can be viewed as a set of net segments. A segment is responsible for making service request and receiving response through a *CP* place. In *F-net*, all segments are connected via some common net elements.

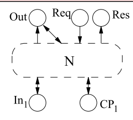
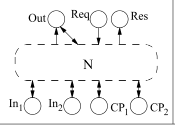
3) CREATING EXOGENOUS CONNECTORS FROM CT-NET

CT-net corresponds to a function (Figure 6) that accepts three input values (for N, O and D) and produces an exogenous connector as output. The exogenous connector is identified by its name (N). Determined by O and D, the behaviour of the produced connector is based on a subset of five functions/operations ($B_N = \{iA, apZ, apL, f_{I-net}, f_{F-net}\}$, where B_N means the behaviour of connector N). In the deployment phase, the behaviour of a connector is based on a subset of four functions (iA, apZ, apL, and f_{I-net}). In the run-time phase, the behaviour of a connector is based on one function (f_{F-net}).



Out
Req
Res
N
O
D
CT-net
N
In$_1$
In$_k$
CP$_1$
CP$_k$
I-net
F-net
where:
A = k,
Initial value of k = { 1, if O=false, 2, otherwise,
|In| = |CP| = A,
Z = {},
L = {},
B$_N$ = { {apL, f$_{I-net}$, f$_{F-net}$}, if O=false ∧ D=false, {apZ, apL, f$_{I-net}$, f$_{F-net}$}, if O=false ∧ D=true, {iA, apL, f$_{I-net}$, f$_{F-net}$}, if O=true ∧ D=false, {iA, apZ, apL, f$_{I-net}$, f$_{F-net}$}, if O=true ∧ D=true.

FIGURE 6. Behaviour of a connector.

	Connector-Net	O	A	N	D	Behaviour (Condition for fixed behaviour)
Unary Connector		false	1	InvC	false	$B_{InvC} = B_{iLoop} = \{apL, f_{I-net}, f_{F-net}\}$
				iLoop		
				Guard	true	$B_{Guard} = B_{fLoop} = \{apZ, apL, f_{I-net}, f_{F-net}\}$ $(Z \neq \{\})$
				fLoop		
Composition Connector		true	2	SEQ	false	$B_{SEQ} = \{iA, apL, f_{I-net}, f_{F-net}\}$
				PIPE	true	$B_{PIPE} = B_{SEL} = \{iA, apZ, apL, f_{I-net}, f_{F-net}\}$
				SEL		

Initially Z=L={}, where 'Z' is the set of constraints and 'L' is the set of selected services for a connector.

FIGURE 7. Exogenous connectors.

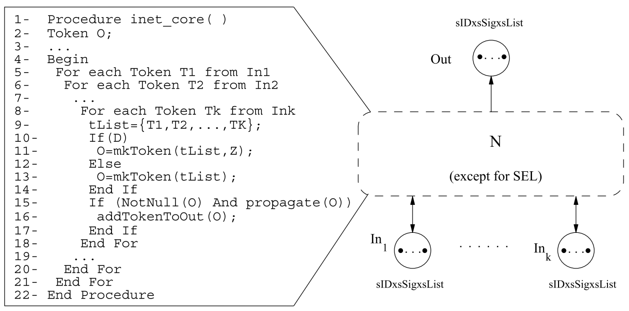
Invoking function iA adds a new input place and a composition place at specific locations along with their handling segments in the respective sub-nets. Functions apZ and apL appends FCL constraints and service IDs to the respective sets in the connector. Function f_{I-net} (to create the component's interface) is triggered by apL, apZ or iA (when a new component is composed by the connector) in the deployment phase. A token value in place *Out*, depending on the respective constraint in Z, is generated by f_{I-net}. Empty set L means that all services are propagated by the connector. Function f_{F-net} represents the behaviour of *F-net* which passes/receives tokens to/from the *CP* places. Function f_{F-net} is triggered by a service request made in the run-time phase.

Deployment phase exogenous connectors created from the CT-net are shown in Figure 7. The set of constraints (Z) and set of selected services (L) is empty for each connector. An exogenous connector (an instance created from CT-net) is a connector-net.

For a system with fixed behaviour, for guard and finite loop connectors, the set of constraints (Z) cannot be empty. In contrast, Z may be empty for the selector and pipe connectors. A pipe without any constraint is a sequencer. A selector without any constraint shows that there is no compound service (no matched services in the composed components).

IV. BEHAVIOUR OF EXOGENOUS CONNECTORS

In this section, the main focus is to describe the core behaviour (f_{I-net} and f_{F-net}) of exogenous connectors at an abstract level in the deployment and run-time phases. Again



```
Procedure inet_core( )
Token O;
...
Begin
 For each Token T1 from In1
  For each Token T2 from In2
   ...
    For each Token Tk from Ink
     tList={T1,T2,...,TK};
     If(D)
      O=mkToken(tList,Z);
     Else
      O=mkToken(tList);
     End If
     If (NotNull(O) And propagate(O))
      addTokenToOut(O);
     End If
    End For
   ...
  End For
 End For
End Procedure
```

FIGURE 8. **Behaviour of I-net.**

adopting top-down approach and by avoiding unnecessary details, we define the behaviour of exogenous connectors schematically and by using high level pseudo code at generic level and then in specific to exogenous connectors.

A. BEHAVIOUR OF I-NET

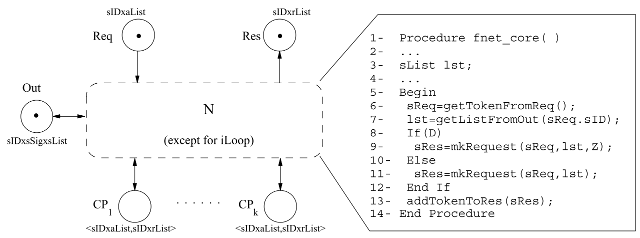
At an abstract level, the behaviour of *I-net* for exogenous connectors (except for the selector connector) is defined in Figure 8. The behaviour of selector's *I-net* is described in Section IV-C. The procedure *mkToken* (a connector specific procedure) creates an output token for each combination of tokens from *In* places. For a constrained connector, the reference of *Z* is passed to overloaded *mkToken*. A token *O* (created by *mkToken*) is then added to the *out* place (by procedure *addTokenToOut*) if the token is not 'null' and represents a selected service in *L* for propagation (checked by procedure *propagate*). All tokens are added to the *out* place if *L* is an empty set ($L = \{\}$).

A token in *In* and *out* places is a tuple ($\langle sID, sSig, sList\rangle$) of service identifier (*sID*), service signature (*sSig*) and an ordered list of sub-services (*sList*) from the connected components. In Figure 8, using CP-net product notation, a colour set inscription (*sIDxsSigxsList*) is shown to *In* and *out* places. The service signature (*sSig*) is a tuple ($\langle sName, oList, iList\rangle$) of service name, lists of output and input parameters. An ordered list of sub-services (*sList*) represents the mapping of the compound service (*sID* from a token in the *out* place) to the services in the *In* places. An element in *sList* is a pair of an *In* place and a service token in the *In* place. The size of *sList* in *Out* is equal to the arity of the connector.

The procedure *inet_core* checks each possible combination of tokens from the input places; however, this does not imply that a token created by *mkToken* is mapped to all or any token in the input places as *mkToken* is exogenous connector specific.

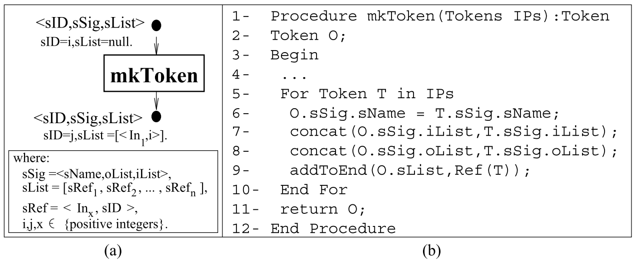
B. BEHAVIOUR OF F-NET

At an abstract level, the behaviour of *F-net* is defined for exogenous connectors (except for the infinite loop connector) by procedure *fnet_core*, as shown in Figure 9. The behaviour of infinite loop's *F-net* is described in Section IV-C.



```
Procedure fnet_core( )
...
sList lst;
...
Begin
 sReq=getTokenFromReq();
 lst=getListFromOut(sReq.sID);
 If(D)
  sRes=mkRequest(sReq,lst,Z);
 Else
  sRes=mkRequest(sReq,lst);
 End If
 addTokenToRes(sRes);
End Procedure
```

FIGURE 9. **Behaviour of F-net.**



```
Procedure mkToken(Tokens IPs):Token
Token O;
Begin
  ...
  For Token T in IPs
   O.sSig.sName = T.sSig.sName;
   concat(O.sSig.iList,T.sSig.iList);
   concat(O.sSig.oList,T.sSig.oList);
   addToEnd(O.sList,Ref(T));
  End For
  return O;
End Procedure
```

FIGURE 10. **Procedure mkToken for the invocation connector.**

On receiving a service request by the connector, the procedure reads the request token from *Req*. The request token is a tuple ($\langle sID, aList\rangle$) of request service ID and a list of arguments for the requested service. Next, the procedure *fnet_core* gets the list of services (referred to as the list of sub-services) for the requested service from *Out* of *I-net*. Next, connector specific procedure *mkRequest* is called by passing the request token and the list of sub-services. For a constrained connector, overloaded *mkRequest* is called by passing additional reference of *Z*. Lastly, procedure *addTokenToRes* adds the response token to the *Res* place of *F-net*.

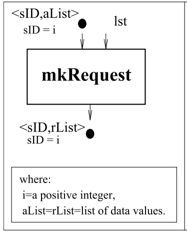
C. CONNECTOR SPECIFIC BEHAVIOUR

In this section, procedures *mkToken* (except for selector) and *mkRequest* (except for the infinite loop) are described for all connectors. *I-net*'s behaviour of the selector and *F-net*'s behaviour for the infinite loop connectors are also defined.

1) INVOCATION CONNECTOR

The procedure *mkToken* accepts a token and returns a token as shown in Figure 10(a). The input token (from the only *In* place) represents a public method of the connected computation unit and the output token (from *mkToken* to *Out*) represents the service exhibited in the interface created by the connector. As methods of the computation unit executes, *sList* of the input token is null. In contrary to this, non-empty *sList* for an output token represents that the connector passes request tokens to the connected component(s).

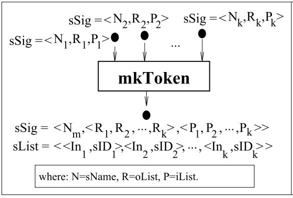
The pseudo code of procedure *mkToken* shown in Figure 10(b) is written to accept a list of input tokens; however, there is one token in the list. The procedure reads the token from the list and copy the details of input token into an output token *O*. Then the procedure adds the reference of the input token (returned by the *Ref* procedure) to the



```
Procedure mkRequest(rqToken T,sList lst):rsToken
rqToken I;
rsToken O;
Integer placeIndex;
Begin
 ...
 For sRef E in lst
  placeIndex = getPlaceIndex(E.pID);
  I = mkReqToken(E.sID,T.aList);
  O = requestTo_CP(placeIndex,I);
 End For
 return O;
End Procedure
```

(a) (b)

FIGURE 11. Procedure mkRequest for the invocation connector.



sSig =<N$_1$,R$_1$,P$_1$>
sSig =<N$_2$,R$_2$,P$_2$>
sSig =<N$_k$,R$_k$,P$_k$>
...
mkToken
sSig = <N$_m$,<R$_1$, R$_2$, ⋯,R$_k$>,<P$_1$, P$_2$, ⋯,P$_k$>>
sList = <<In$_1$,sID$_1$>,<In$_2$,sID$_2$>, ⋯, <In$_k$,sID$_k$>>
where: N=sName, R=oList, P=iList.

```
Procedure mkToken(Tokens IPs):Token
Token O;
Begin
 ...
 For each  Token T in IPs
  {O.sSig.sName=T.sSig.sName;}
  concat(O.sSig.iList,T.sSig.iList);
  concat(O.sSig.oList,T.sSig.oList);
  addToEnd(O.sList,Ref(T));
 End For
 return O;
End Procedure
```

(a) (b)

FIGURE 12. Procedure mkToken for sequencer.

output token. An element in *sList* is a pair of a place reference and a service reference (from the input token) in the place. This information is needed by the invocation connector in the run-time phase for service execution, as shown in Figure 11. Lastly, the procedure returns the created token.

The procedure *mkRequest* accepts two arguments (a request token and a list of sub-services of the requested service) and returns a response token. For the invocation connector, the argument *lst* has exactly one sub-service. Request token *T* is a pair of requested service ID and a list of arguments for the requested service. Procedure *mkRequest* creates a request token *I* (for the sub-service in the *lst* argument) and sends token *I* to the respective *CP* place by calling *requestTo_CP*. Lastly, the response of the service execution is returned by the procedure.

As a sample, connector net of invocation connector is simulated by using CPN tools, as described in Section V-A. In contrast with the CPN model, it is easier to follow the pseudo code presentation.

2) SEQUENCER

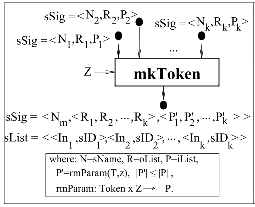
Procedure *mkToken* of sequencer creates a combined (or compound) token from the two or more input tokens; *mkToken* (Figure 12(b)) is a modified version (modifications are shown in bold) of *mkToken* from Figure 10.

In Figure 12(a), for simplicity, only *sSig* of input and output tokens, and *sList* of the output token are shown. In an output token, the service signature contains ordered lists of input/output parameters of the service signatures from the input tokens. The output token contains a list (sList) of tuples. The mechanism for naming a compound service is not shown.

Pseudo code of the invocation connector's *mkRequest* (Figure 11) is modified for sequencer, as shown in Figure 13. The argument *lst* to the procedure *mkRequest* of the sequencer connector has two or more tokens. The modified

```
Procedure mkRequest(rqToken T,sList lst):rsToken
rqToken I;
rsToken O,tmp;
Integer placeIndex,bgIndex,paramSize;
Begin
 ...
 For each sRef E in lst
  placeIndex=getPlaceIndex(E.pID);
  paramSize=getParamSize(E.sID);
  I=mkReqToken(E.sID, subLst(T.aList,bgIndex,paramSize));
  bgIndex=bgIndex+paramSize;
  tmp=requestTo_CP(placeIndex,I);
  appendToList(O.rList,tmp.rList);
 End For
 return O;
End Procedure
```

FIGURE 13. Procedure mkRequest for sequencer.



sSig =<N$_1$,R$_1$,P$_1$>
sSig =<N$_2$,R$_2$,P$_2$>
sSig =<N$_k$,R$_k$,P$_k$>
...
Z
mkToken
sSig = <N$_m$,<R$_1$, R$_2$, ⋯,R$_k$>,<P'$_1$, P'$_2$, ⋯,P'$_k$ >>
sList = <<In$_1$,sID$_1$>,<In$_2$,sID$_2$>, ⋯, <In$_k$,sID$_k$>>
where: N=sName, R=oList, P=iList,
P'=rmParam(T,z), $|P'| \leq |P|$,
rmParam: Token x Z⟶ P.

```
1-Procedure mkToken(Tokens IPs,Z z ):Token
2-Token O;
3-iList lst;
4-Begin
5-   ...
6-  For each Token T in IPs
7-   lst=rmParam(T,z);
8-   concat(O.sSig.iList, lst );
9-   concat(O.sSig.oList,T.sSig.oList);
10-  addToEnd(O.sList,Ref(T));
11- End For
12- return O;
13-End Procedure
```

(a) (b)

FIGURE 14. Procedure mkToken for pipe.

mkRequest procedure finds out the arguments for each sub-service from the *lst* argument, makes request to the sub-service and appends the response of the sub-service into the response token *O*. After getting the response of the last sub-service, the procedure returns the response token.

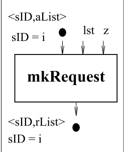
A simplified CPN model of the sequencer connector is described in Section V-B and behaviour of a sequencer connector SEQ1 is illustrated in Section IV-D with the help of a practical example.

3) PIPE

Procedure *mkToken* of pipe (Figure 14) is a modified version of sequencer's *mkToken* (Figure 12). Input parameters for a service taken from other services' result sets (defined in *z*) are not appended to the output token's *iList*, as shown in Figure 14(a); this is achieved by statements 7 and 8 in Figure 14(b). For a token, *rmParam* returns a list of input parameters which are not *rpID* in the respective constraint.

The procedure *mkRequest* (Figure 15(a)) accepts a request token, a list of sub-services of the requested service and the set of constraints, and returns a response token. The procedure *mkRequest* of pipe (Figure 15(b)) is the modified version of sequencer's *mkRequest* (Figure 13).

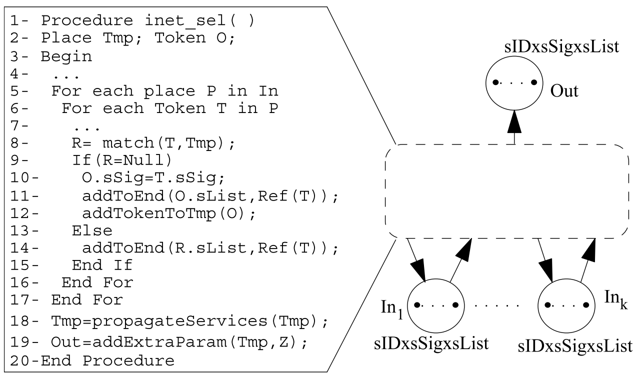
In contrast with sequencer, pipe's *mkRequest* gets the argument values either from the result data of other sub-services or from the argument list of the requested service, as shown in Figure 15(b) on code lines 10-18. Procedure *inSet* checks if there is a tuple in the first argument (*tmpList*) that uses a certain parameter *P* of a service in *E* as a receiver of a value. Procedure *getValue* reads a response value of the sender service which is used as a specific parameter *P* of the service in *E*. The rest of the behaviour is the same as with sequencer. The behaviour of pipe connectors PIPE1 and



```
1-Procedure mkRequest(rqToken T,sList lst, Z z ):rsToken
2-rqToken I;rsToken O,tmp;
3- Integer placeIndex,bgIndex; {paramSize;}
4-iList pList;Token S;...;
5-Begin
6-   ...
7-   For each sRef E in lst
8-    placeIndex=getPlaceIndex(E.pID);
9-    {paramSize=getParamSize(E.sID);}tmpList=translate(z);
10-   S=getServiceToken(E.sID);
11-   For each Param P in S.sSig.iList
12-    If(Not inSet(tmpList,E.sID,P))
13-     append(pList,T.aList[bgIndex]);
14-     bgIndex=bgIndex+1;
15-    Else
16-     append(pList,getValue(tmpList,E.sID,P));
17-    End IF
18-   End For
19-   I=mkReqToken(E.sID, pList);
20-    {bgIndex=bgIndex+paramSize;}
21-   tmp=requestTo_CP(placeIndex,I);
22-   appendToList(O.rList,tmp.rList);
23- End For
24- return O;
25-End Procedure
```

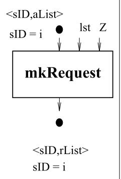
(a) (b)

FIGURE 15. Procedure mkRequest for pipe.



```
1- Procedure inet_sel( )
2- Place Tmp; Token O;
3- Begin
4-   ...
5-   For each place P in In
6-    For each Token T in P
7-     ...
8-     R= match(T,Tmp);
9-     If(R=Null)
10-     O.sSig=T.sSig;
11-     addToEnd(O.sList,Ref(T));
12-     addTokenToTmp(O);
13-    Else
14-     addToEnd(R.sList,Ref(T));
15-    End If
16-   End For
17- End For
18- Tmp=propagateServices(Tmp);
19- Out=addExtraParam(Tmp,Z);
20-End Procedure
```

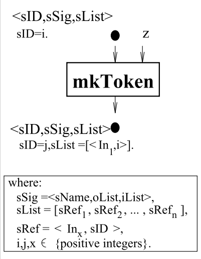
FIGURE 16. I-net for selector.



```
1-   Procedure mkRequest(rqToken T,sList lst,Z z ):rsToken
2-   ...
3-   Begin
4-    ...
5-    {For each sRef E in lst}
6-    If(sizeOf(lst)>1)
7-     componentRef=evaluate(z,T);
8-     E=getRef(lst,componentRef);
9-    Else
10-    E=lst[0];
11-   End If
12-   placeIndex=getPlaceIndex(E.pID);
13-   paramSize=getParamSize(E.sID);
14-   I=mkReqToken(E.sID,subLst(T.aList,bgIndex,paramSize));
15-   { bgIndex=bgIndex+paramSize;}
16-   tmp=requestTo_CP(placeIndex,I);
17-   appendToList(O.rList,tmp.rList);
18-   {End For}
19-   return O;
20- End Procedure
```

(a) (b)

FIGURE 17. Procedure mkRequest for selector.



```
1-   Procedure mkToken(Tokens IPs ,Z z):Token
2-   Token O;
3-   Begin
4-    ...
5-    For Token T in IPs
6-     If(isIn(T,z))
7-      O.sSig.sName=T.sSig.sName;
8-      concat(O.sSig.iList,T.sSig.iList);
9-      concat(O.sSig.oList,T.sSig.oList);
10-     addToEnd(O.sList,Ref(T));
11-     If(extraParam(z,O))
12-      addExtraParam(z,O));
13-     End If
14-     markParamConstrained(O.sSig.oList);
15-    End If
16-   End For
17-   return O;
18- End Procedure
```

(a) (b)

FIGURE 18. Procedure mkToken for guard.

PIPE2 are illustrated in Section IV-D with the help of a practical example.

4) SELECTOR

The behaviour of selector's I-net is shown in Figure 16. For each token of each *In* place, the procedure *inet_sel* adds the token (pointing to the token of the *In* place) in *Tmp* if the token is not found in *Tmp*; otherwise, the current token's reference from *In* is appended to the found token from *Tmp*. After processing all tokens, the procedure *propagateServices* removers service tokens from *Tmp* that are not in '*L*' and procedure *addExtraParam* adds any extra parameters to the service signatures from the respective constraints. Finally, place *Tmp* is assigned to place *Out*.

The procedure *mkRequest* for the selector connector (Figure 17) is a modified version of sequencer's *mkRequest* (Figure 13). The procedure *mkRequest* creates a request to one sub-service from the *lst* argument. For a compound service (*lst* has more than one elements), the procedure selects a reference from *lst* by evaluating the concerned constraint; alternatively, the first reference from *lst* is selected. The element *aList* in the request token may be containing extra values for selection decision; hence, the actual input parameter size is found out on line 13. Next, the procedure makes the request and returns the result set into the response token. The behaviour of a selector connector is illustrated in Section IV-D.

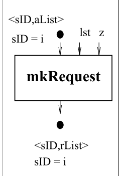
5) GUARD CONNECTOR

As with selector, constraints of a guard can also add extra parameters in a service's signature. The procedure *mkToken* of the guard connector (Figure 18) is a modified version of procedure *mkToken* of the invocation connector (Figure 10). Procedure *isIn* checks for the constraint of the first parameter's service in the second parameter.

The modified procedure accepts two inputs and creates an output token for a constrained service. If a service's constraint is using some parameters other than the service's input parameters (checked by procedure *extraParam*), procedure *addExtraParam* adds more input parameters to the service's signature. Before returning the token, procedure *markParamConstrained* marks each output parameter of the token 'O' as constrained.

For guard, procedure *mkRequest* (Figure 19(b)) is a modified version of procedure *mkRequest* of the invocation connector (from Figure 11).

Initially, the modified procedure *mkRequest* creates null values for all output parameters for the requested service and evaluates the requested service's constraint in '*z*'. If the guard condition is satisfied, then the procedure forwards the service request. The behaviour of three different guard connectors is illustrated in Section IV-D.

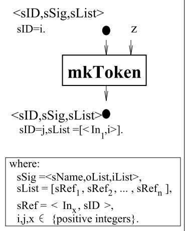


(a)

```
Procedure mkRequest(rqToken T,sList lst,Z z ):rsToken
rqToken I;
rsToken O;
Integer placeIndex, bgIndex,paramSize;
Begin
 ...
 For sRef E in lst
  paramSize=getParamSize(E.sID);
  genNullValues(O,paramSize);
  If(evaluate(T,z))
    placeIndex=getPlaceIndex(E.pID);
    I=mkReqToken(E.sID, subLst(T.aList,bgIndex,paramSize));
    O=requestTo_CP(placeIndex,I);
  End If
 End For
 return O;
End Procedure
```

(b)

FIGURE 19. Procedure mkRequest for guard.



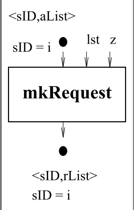
<sID,sSig,sList>
sID=i.
z
mkToken
<sID,sSig,sList>
sID=j,sList =[< In$_1$,i>].
where:
sSig =<sName,oList,iList>,
sList = [sRef$_1$, sRef$_2$, ... , sRef$_n$],
sRef = < In$_x$, sID >,
i,j,x ∈ {positive integers}.

(a)

```
Procedure mkToken(Tokens IPs ,Z z ):Token
Token O;
Begin
 ...
 For Token T in IPs
  If(isIn(T,z))
   O.sSig.sName=T.sSig.sName;
   concat(O.sSig.iList,T.sSig.iList);
   concat(O.sSig.oList,T.sSig.oList);
   addToEnd(O.sList,Ref(T));
  End If
 End For
 return O;
End Procedure
```

(b)

FIGURE 20. Procedure mkToken for the finite loop connector.



(a)

```
Procedure mkRequest(rqToken T,sList lst,Z z ):rsToken
rqToken I;
rsToken O,  tmp;
Integer placeIndex, count;
Begin
 ...
 For sRef E in lst
  placeIndex=getPlaceIndex(E.pID);
  I=mkReqToken(E.sID,T.aList);
  Repeat
   tmp =requestTo_CP(placeIndex,I);
   count=count+1;
  Until(evaluateTermination(E,z,tmp,count)
  appendToList(O.rList,tmp.rList);
 End For
 return O;
End Procedure
```

(b)

FIGURE 21. Procedure mkRequest for the finite loop connector.

6) FINITE LOOP CONNECTOR

The procedure *mkToken* of the finite loop connector (Figure 20) is a modified version of procedure *mkToken* of the invocation connector (Figure 10).

The procedure *mkRequest* of the finite loop connector (Figure 21) is a modified version of the invocation connector's *mkRequest* (Figure 11).

In the modified procedure, for executing the request, the procedure enters in a loop construct. After completing an iteration, procedure *evaluateTermination* evaluates the constraint for loop termination. After the termination of this loop, the response token (*tmp*) is appended to *O* for return. The behaviour of a finite loop connector L1 is illustrated in Section IV-D in the bank example.

7) INFINITE LOOP CONNECTOR

The infinite loop connector is a unary connector without any constraint; the connector will keep sending control to the adapted component forever. Such a connector is used as a root connector when the system is completed.

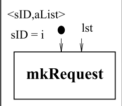
(a)

```
...
  For Token T in IPs
   O.sSig.sName=T.sSig.sName;
   {concat(O.sSig.iList,T.sSig.iList);}
   {concat(O.sSig.oList,T.sSig.oList);}
   addToEnd(O.sList,Ref(T));
...
```

(b)

```
...
Begin
 sReq=getTokenFromReq();
 lst=getListFromOut(sReq.sID);
 mkRequest(sReq,lst);
End Procedure
```

FIGURE 22. Procedures mkToken and fnet_core for the infinite loop connector.



(a)

```
Procedure mkRequest(rqToken T,sList lst){,Z z):rsToken}
rqToken I;
{rsToken O,tmp;}
Integer placeIndex; {count;}
Begin
 ...
 For sRef E in lst
  placeIndex=getPlaceIndex(E.pID);
  I=mkReqToken(E.sID,T.aList);
  Repeat
   tmp=requestTo_CP(placeIndex,I);
   {count=count+1}
 Until(true)    {(evaluateTermination(E,z,tmp,count)}
   {appendToList(O.rList,tmp.rList);}
 End For
 {return O;}
End Procedure
```

(b)

FIGURE 23. Procedure mkRequest for the infinite loop connector.

The procedure *mkToken* for the infinite loop connector (Figure 22(a)) is the modified version of the procedure *mkToken* of the invocation connector (Figure 10). The modifications show that the infinite loop connector only copy the name of the input token to the output token. A service token generated by the infinite loop connector does not show the input/output parameters of the service.

Procedure *fnet_core* for other exogenous connectors (shown in Figure 9) is modified for the infinite loop connector, as shown in Figure 22(b). No response token is produced by this procedure.

The *mkReguest* procedure for the infinite loop (Figure 23) is a simplified and modified version of the *mkReguest* procedure for the finite loop connector (Figure 21). In the procedure, the connector repeats sending request token to its *CP* place infinitely. For the infinite loop connector, the response token is never sent to the *Res* place. The behaviour of an infinite loop connector L2 is described in Section IV-D in the bank example.

D. THE BANK EXAMPLE

In this section, an extended system (shown in Figure 24) of the bank example from Section II is used to illustrate the behaviour of exogenous connectors. Service interfaces of basic and composite components and FCL constraints of constrained connectors are shown in Figure 24. Service names are not changed by unary connectors; however, service names for the composite connectors can be renamed meaningfully by the developers.

The composite of two components CR and PR (both with one service) are composed by a sequencer connector SEQ1; the compound service is named *getData* which executes the sub-services *readCard* of CR and *readPin* of PR. The input and output of this compound service is the concatenation lists of inputs and outputs of the sub-services as described in Section IV-C2. This composite of SEQ1 is further composed with CB component for card authentication by using the PIPE1 connector. The two output parameters of SEQ1 are passed as input parameters to CB as defined by the constraint of PIPE1. The composite of PIPE1 is adapted by a finite loop connector L1; L1 does not change service interface as described in Section IV-C6. Constraint of L1 iterates maximum three times with a condition to terminate on successful card authentication.

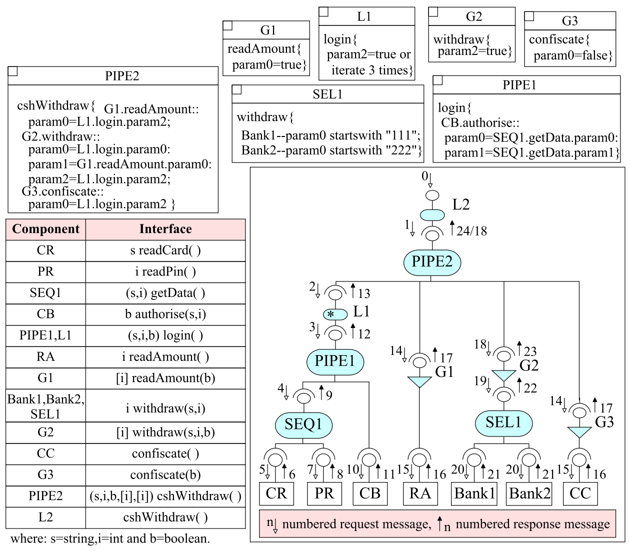
Selector SEL1 composes two components for the bank branches served by the machine; based on the details of card, a *withdraw* service request is routed to one component only. The composite of SEL1 is adapted by guard connector G2; defined by the constraint, G2 sends the request if the card is authenticated. RA is component to accept the amount to withdraw; this component is adapted by guard connector G1. defined by the constraint, G1 sends the request if the card is authenticated. A component CC to confiscate the card is adapted by guard connector G3; defined by the constraint, G3 sends the request if the card is not authenticated. Connectors G1, G2 and G3 changes the service signature as described in Section IV-C5.

Connector PIPE2 composes four adapted components of L1, G1, G2 and G3. In PIPE2 constraint, the successful card authentication data is passed to service request to G1 adapted component for reading amount to withdraw. The withdraw mount along with card data and successful card authentication result is passed to G1 adapted component for amount withdrawal. For unsuccessful card authentication result is passed to G3 adapted component to confiscate the card.

PIPE2 component is adapted by an indefinite loop connector L2. During system execution, after serving first customer request, L2 iterates and sends the control to CR component to read card for next customer service. For a service adopted by indefinite loop connector, there is no input or output parameters as described in Section IV-C7. The numbered arrows shown in Figure 24 represent the sequence of requests and responses through the system at the run-time. For a request, a response is generated by each connector except the root connector.

V. CPN MODELS OF SAMPLE EXOGENOUS CONNECTORS

In order to verify the specifications of exogenous connectors created from our proposed pattern, we have created CPN models for all connectors. For each connector, we have created two models to verify the design-time behaviour (as defined by the I-net) and the run-time behaviour (as defined by the F-net). The purpose of creating these models and their simulations is not to define the generic behaviour of exogenous connectors, but to illustrate that the CT-net based


G1
readAmount{ param0=true}
L1
login{ param2=true or iterate 3 times}
G2
withdraw{ param2=true}
G3
confiscate{ param0=false}
PIPE2
cshWithdraw{ G1.readAmount:: param0=L1.login.param2; G2.withdraw:: param0=L1.login.param0: param1=G1.readAmount.param0: param2=L1.login.param2; G3.confiscate:: param0=L1.login.param2 }
SEL1
withdraw{ Bank1--param0 startswith "111"; Bank2--param0 startswith "222"}
PIPE1
login{ CB.authorise:: param0=SEQ1.getData.param0: param1=SEQ1.getData.param1}
0
L2
1
24/18
PIPE2
2
13
L1
3
12
PIPE1
14
17
G1
18
23
G2
19
22
4
9
SEQ1
SEL1
14
17
G3
5
6
7
8
10
11
15
16
20
21
20
21
15
16
CR
PR
CB
RA
Bank1
Bank2
CC
n numbered request message, n numbered response message

Component	Interface
CR	s readCard( )
PR	i readPin( )
SEQ1	(s,i) getData( )
CB	b authorise(s,i)
PIPE1,L1	(s,i,b) login( )
RA	i readAmount( )
G1	[i] readAmount(b)
Bank1,Bank2, SEL1	i withdraw(s,i)
G2	[i] withdraw(s,i,b)
CC	confiscate( )
G3	confiscate(b)
PIPE2	(s,i,b,[i],[i]) cshWithdraw( )
L2	cshWithdraw( )

where: s=string,i=int and b=boolean.

FIGURE 24. The extended bank system.

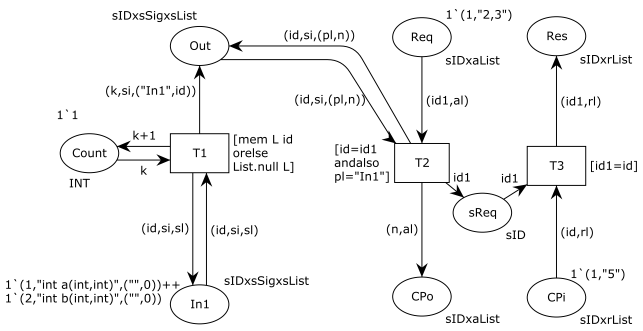


FIGURE 25. InvC CPN model structure.

definition (from Section III) and the abstract level behaviour of exogenous connectors in Section IV are achievable.

Here, we model and simulate invocation connector amongst the unary connectors and sequencer connector (with simplifications) amongst the composition connectors.

A. INVOCATION CONNECTOR

Using CPN tools, a CP-net of the invocation connector (*InvC-net*) is shown in Figure 31 (the CPN model structure is shown in Figure 25); this net models/simulates the interface generation of a component and the invocation of a service in the run-time phase. *I-net* of *InvC-net* defines how the invocation connector on connection with a computation unit (place *In1* of *I-net*) creates the interface of an atomic component (place *Out* of *I-net*). *F-net* of *InvC-net* defines how the connector forwards a service request to a computation unit and then returns the execution results as the service response.

To simulate interface generation, a computation unit with two public methods (method 'a' computes addition of two numbers and method 'b' computes difference of two number) is connected; two tokens are added in the *In1* place of *I-net*. Similarly, to simulate method invocation from the

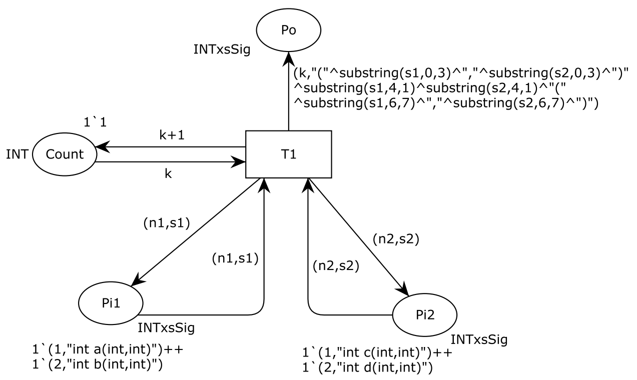


FIGURE 26. Structure of I-net of sequencer connector.

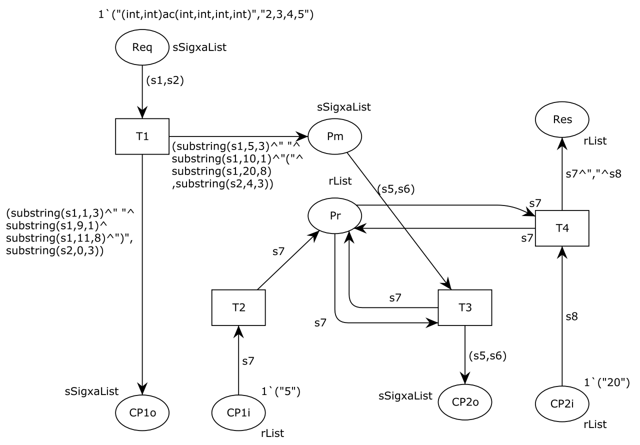


FIGURE 27. F-net before simulation.

connected computation unit, a service request token is added in the *Req* place and a result token of a method invocation is added in the *CP1* place. In this net, a list variable *L* with empty list is used to represent element *L* of CT-net; by default, *L* is empty. CP-net does not support adding a function *apL* to be invoked directly in a net; hence, to achieve the behaviour of *apL*, service IDs are entered in list *L* manually.

For simulation, in CPN tools, we choose an option to execute a transition with chosen binding. Initially, in *InvC-net*, transition *T1* is enabled. Selecting the first method from the *In1* place creates a new service for the atomic component with an *sID* based on current value of *k* in the *Count* place. Now transition *T2* becomes enabled. In the same way, the second method from the *In1* place is selected. Currently, there are two services in the *Out* place, as shown in Figure 32.

Next, in order to simulate *InvC-net*'s operation in the run-time phase, transition *T2* is triggered. *T2* passes the argument data from the request token and the method's *id* (from the *Out* place matched with the requested service) to the *CPo* place. After execution of *T2*, transition *T3* becomes enabled. Lastly, *T3* is triggered to return the response token by reading the executed method's results from *CPi* if the executed method's id is matched with *sID* from the *sReq* place.

B. SEQUENCER CONNECTOR

For clarity, CNP models of the sequencer connector are further simplified. Firstly, models of *I-net* (Figure 33) and *F-net* (Figure 34) for the sequencer connector are created separately.

As the focus of these models is to simulate the interface generation and the flow of control/data to the composed components, in the *I-net*, types of *In* and *Out* places do not show *sList*. In the *F-net*, as the requested service is not checked from the *Out* place, instead of service ID (*sID*), service signature (*sSig*) is used in the type of concerned places. Hence, *sID* is not included in the response token. In the *I-net*, tokens in *In1* represent addition and subtraction services for two numbers. Tokens in *In2* represent multiplication and division services for two numbers. As with *apL*, behaviour of function *iA* can also be achieved manually by incrementing the arity variable and by extending the two nets for the newly added component.

The structure of CPN model of the I-Net of Sequencer connector is shown in Figure 26. For simulation, in CPN tools, we choose an option to execute a transition with chosen binding. From the *In* places, selecting tokens correctly, four possible combined services are generated, as shown in the *Out* place in Figure 33.

The *F-net* of the sequencer (structure is shown in Figure 27) is simplified as mentioned earlier. Initially, transition T1 and T2 are enabled, as shown in Figure 34. Request token in *Req* place corresponds to a combined service 'ac'. This token holds data values for the requested service. A token in *CP1i* represents the execution result (addition of two numbers) of service 'a' for data values ('2' and '3') in the request token. Similarly, token in *CP2i* represents the execution result (multiplication of two numbers) of service 'c' for data values ('4' and '5') in the request token.

In the *F-net*, for correct simulation, transitions should be triggered in sequence *T1*, *T2*, *T3* and *T4*. The result of two services ('a' and 'c') is added as a tuple of two values ('5' and '20') in the *Res* place, as shown in Figure 35.

VI. EXOGENOUS COMPOSITION FRAMEWORK

In order to validate the methodology, we have implemented the exogenous connector of EX-MAN in a prototype tool exogenous composition framework (ECF). These connectors are used in building many component based systems [39], [41] and the overall behaviour of connectors in the system is checked and simulated. To model and simulate EX-MAN systems with the defined exogenous connectors, we have developed a prototype tool (Figure 28(a)) ECF. In this tool, exogenous connectors (described in Section IV) are implemented and reused through the connector repository. In order to demonstrate the usefulness of this tool, a simple composite 'sample' is shown by using encapsulated

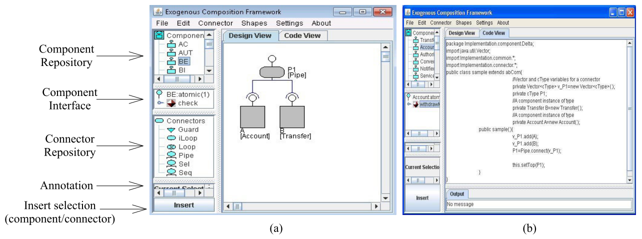


FIGURE 28. Visual construction environment of ECF [38].

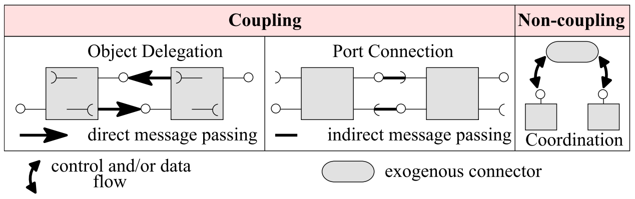


FIGURE 29. Connectors for software composition.

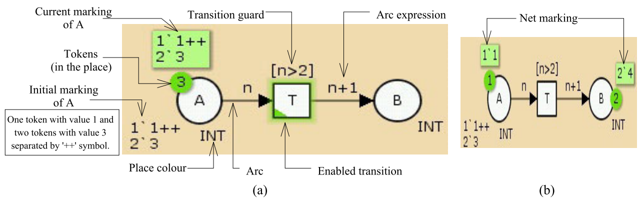


FIGURE 30. A model in CPN tools before and after simulation.

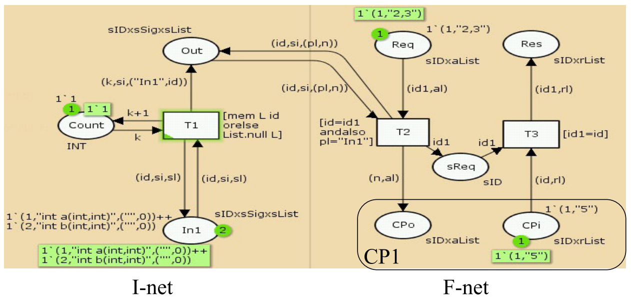


FIGURE 31. InvC CPN model before simulation.

components and the connectors from the repositories of components and connectors. The code of the composite can be seen by switching to the code view tab (Figure 28(b)) of the tool. With the help of this tool, the partial systems during the construction process and the final system are tested by simulation of the system with test data. In textual form, the results of simulations are shown in the output tab of the tool. The bank example from Section IV-D is constructed and simulated in ECF.

VII. RELATED WORK

For system construction, a composition mechanism [26] puts two or more programs units to create a composite program unit. In general, there are three kinds of composition connectors (shown in Figure 29) for composing components in CBD [26]. In contrast with other mechanisms (e.g. port connections in architectural description languages or object delegation in object-oriented languages) coordination does not induce coupling between the composed program units. Coordination based composition is defined as independent program units in Reo, X-MAN and EX-MAN. Hence, coordination is the most appropriate composition mechanism for system construction in CBD; this is also in line with the programming-in-the-large concepts [8].

Coordination models and languages are categorised into two groups in [34]: data-driven and control-driven. The examples of data coordination are using tuple spaces [7] and data connectors [3] for parallel processes or active components. Using orchestration [10] for (web) services and exogenous composition connectors in EX-MAN are examples of control coordination. Coordination based composition concept is used in Reo, X-MAN, EX-MAN and used for web service composition. One benefit of our proposed pattern is the availability of a common framework for coordination based composition. In CBD, there are many component models in which connectors are defined as separate entities than components. In this section, we briefly mention few such examples.

In Acme [13], matching ports of two components are connected by one of many connectors. Few examples of such connectors are message passing (or procedure call) connectors, event broadcasting connectors, database queries connectors and pipes [14]. In C2 (from Chiron-2 [46]), message passing devices or connectors are referred to as bus and concurrent components are composed by these devices. A bus broadcasts requests/notifications to its connected components. In a system design by SOFA 2.0 component model [36], the communication between components takes place with the invocation of a method through the connected ports of the components.

In Reo [3], defined active components are connected to communicate via channels (streams) for data exchange. Representing coordination data patterns, there are many different kinds of basic and composite channels in Reo. New composite connectors can always be created from the existing connectors. Composing web services [1] by coordination is referred to as orchestration [11], [15], [34]; the composite of this process produces workflows. BPEL is a well known language [33] to implement orchestration for web service composition. In such a composite, participating web services are separated from the orchestrating workflow. Orchestration for the composition of web services are not predefined connectors but glue code written as per the requirement to compose two specific services.

For the higher re-usability, the idea of dealing with the computation and control is widely adopted in the software development community [2], [30], [35], [48]. In order to separate the control from computation and further division of control into manageable smaller units for reuse, the concept of connector is getting wider acceptability. In the

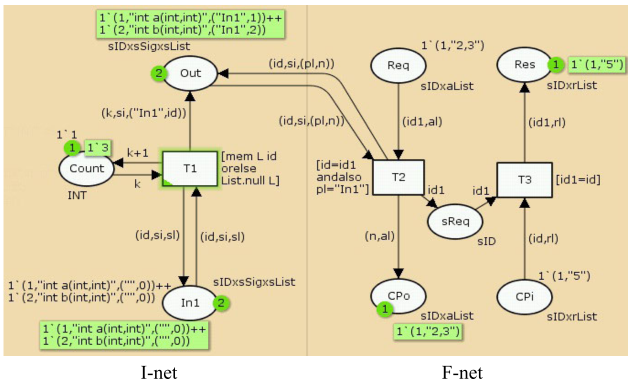
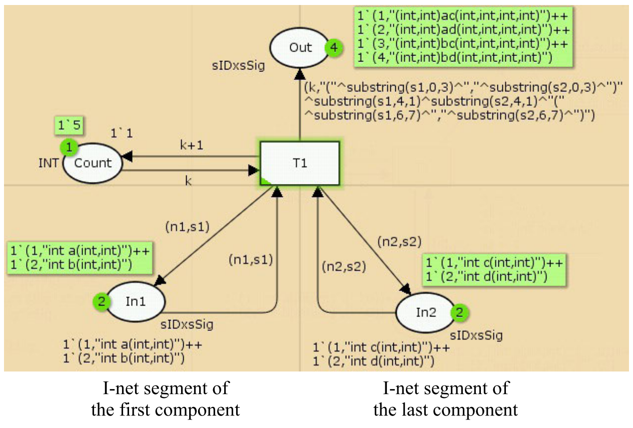
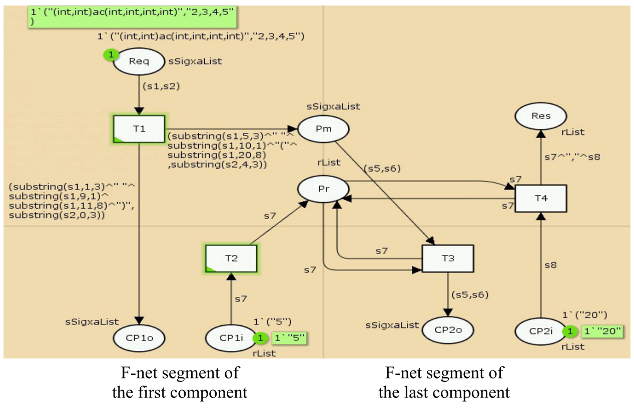


FIGURE 32. InvC CPN model after simulation.



I-net segment of the first component
I-net segment of the last component

FIGURE 33. I-net.



F-net segment of the first component
F-net segment of the last component

FIGURE 34. F-net before simulation.

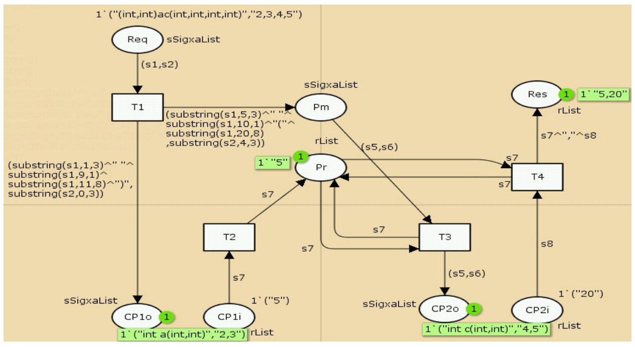


FIGURE 35. Transitions triggered in sequence *T1*, *T2*, *T3* and *T4*.

common component architecture (CAA) model proposed in [2], the interactions between components are handled by two different kind of connectors: endogenous connectors and exogenous connectors. These connectors are independent program units to define the interaction patterns between the connected components. Similarly, PUTRACOM component model [31] adopts the exogenous connector as third party units with further extensions for the concurrent processes handling.

VIII. DIRECTIONS FOR FUTURE WORK

The separation of control and computation into two layers in EX-MAN is better than a flat architecture of components and connectors created in other component models. However, the hierarchies of exogenous connectors into many levels leads to a complex structure. The complexity of this control structure can be reduced if some part of this structure (containing more than one connectors) is replaced by one composite connector. To reduce this hierarchy of connectors systematically, a careful investigation is required which can lead to find new composite connectors for reuse. Such composite connectors can be stored and reused through the connector repository.

For fixing the behaviours of exogenous connector in EX-MAN, there are a number of limitations in FCL. Currently, EX-MAN and its connectors are used to defined sequential system construction. In future, we would like to extend the capability of EX-MAN so that concurrent systems can be built. This extension would require to revisit the definition of our proposed pattern. In this regard, we intend to explore the component model with concurrency from [32].

Another direction for the extension of our work is to investigate the possibility of defining the communication/coordination patterns in the design patterns defined in [12]. Our work in [24] is the motivation for this direction in the future work. We would also like to investigate the possibility of enabling our model to be able to generate as many connectors possible from [48].

Keeping in view the aforesaid future developments, the support of a suitable tool is always desired. Hence, we would like to investigate the future extensions and development of the ECF. We intend to investigate ways to automate the refinements of connector constraints for modification of a system. This will help the system developers to save a considerable amount of development/maintenance time.

APPENDIX
SAMPLE NETS IN CPN TOOLS

A simple CPN model is shown in CPN tools before and after simulation in Figure 30.

The CPN model for invocation connector before and after simulations are shown in Figure 31 and Figure 32, respectively.

The I-Net CPN model for sequencer connector is shown in Figure 33.

The CPN model for F-Net of Sequencer connector before and after simulations are shown in Figure 34 and Figure 35, respectively.

ACKNOWLEDGMENT

The authors would like to thank the Deanship of Scientific Research at Umm Al-Qura University for supporting this work by grant code 19-COM-1-01-0015.

REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Berlin, Germany: Springer, 2004.

[2] G. A. Araájo, F. H. Carvalho, and R. C. Corráa, "Implementing endogenous and exogenous connectors with the common component architecture," in *Proc. Workshop Component-Based High Perform. Comput.*, 2009, pp. 1–4.

[3] F. Arbab, "Reo: A channel-based coordination model for component composition," *Math. Struct. Comput. Sci.*, vol. 14, no. 3, pp. 329–366, Jun. 2004.

[4] D. Arellanes and K.-K. Lau, "Algebraic service composition for user-centric iot applications," in *Internet Things*, D. Georgakopoulos and L.-J. Zhang, Eds. Cham, Switzerland: Springer, 2018, pp. 56–69.

[5] X. Bellekens, R. Atkinson, A. Seeam, C. Tachtatzis, I. Andonovic, and K. Nieradzinska, "Cyber-physical-security model for safety-critical iot infrastructures," in *Wireless World Research Forum Meeting*, vol. 35, p. 8, Oct. 2016.

[6] B. Meyer and K. Arnout, "Componentization: The visitor example," *Computer*, vol. 39, no. 7, pp. 23–30, Jul. 2006.

[7] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[8] F. DeRemer and H. H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Trans. Softw. Eng.*, vols. SE–2, no. 2, pp. 80–86, Jun. 1976.

[9] J. Desel and W. Reisig, "The concepts of Petri nets," *Softw. Syst. Model.*, vol. 14, no. 2, pp. 669–683, May 2015.

[10] T. Erl, *Service-Oriented Architecture: Concepts, Technology Design*. Upper Saddle River, NJ, USA: Prentice-Hall, 2005.

[11] J. Fiadeiro, A. Lopes, and L. Bocchi, "A formal approach to service component architecture," in *Proc. 3rd Int. Workshop Services Formal Methods*. Berlin, Germany: Springer, 2006, pp. 193–213.

[12] E. Gamma, J. Vlissides, R. Johnson, and R. Helm, *Design Patterns CD: Elements of Reusable Object-Oriented Software (CD-ROM)*. Boston, MA, USA: Addison-Wesley, 1998.

[13] D. Garlan, R. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," *Foundations Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge, U.K.: Cambridge Univ. Press, 2000, pp. 47–68.

[14] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-94-166, 1994.

[15] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, p. 96, Feb. 1992.

[16] A. Hannan, S. Arshad, M. Azam, J. Loo, S. Ahmed, M. Majeed, and S. Shah, "Disaster management system aided by named data network of things: Architecture, design, and analysis," *Sensors*, vol. 18, no. 8, p. 2431, Jul. 2018.

[17] E. E. Hayek, I. G. Ben Yahia, D. Arellanes, and K.-K. Lau, "Analysis of component-based approaches toward componentized 5G," in *Proc. 21st Conf. Innov. Clouds, Internet Netw. Workshops (ICIN)*, Feb. 2018, pp. 1–5.

[18] G. Heineman and W. Councill, *Component-Based Software Engineering*. Boston, MA, USA: Addison-Wesley, 2001.

[19] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, vol. 2. London, U.K.: Springer-Verlag, 1995.

[20] K. Jensen, "A brief introduction to coloured Petri nets," in *Proc. 3rd Int. Workshop Tools Algorithms Construct. Anal. Syst.*, London, U.K.: Springer-Verlag, 1997, pp. 203–208.

[21] K.-K. Lau and S. Cola, *An Introduction to Component-Based Software Development*. Singapore: World Scientific, 2017.

[22] K.-K. Lau, "Towards composing software components in both design and deployment phases," in *Proc. 10th Int. Symp. Compon.-Bsased Softw. Eng.*, 2007, pp. 274–282.

[23] K.-K. Lau, L. Ling, P. V. Elizondo, and V. Ukis, "Composite connectors for composing software components," in *Proc. 6th Int. Symp. Softw. Composition*, M. Lumpe and W. Vanderperren, Eds. Berlin, Germany: Springer-Verlag, 2007, pp. 266–280.

[24] K.-K. Lau, I. Ntalamagkas, C. Tran, and T. Rana, "Design patterns as composition operators," in *Proc. 13th Int. Symp. Compon.-Based Softw. Eng.*, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer-Verlag, 2010, pp. 232–251.

[25] K.-K. Lau, M. Ornaghi, and Z. Wang, "A software component model and its preliminary formalisation," in *Proc. 4th Int. Symp. Formal Methods Compon. Objects*, F. S. de BoerMarcello, M. BonsangueSusanne, and G.-P. de Roever, Eds. Berlin, Germany: Springer-Verlag, 2006, pp. 1–21.

[26] K.-K. Lau and T. Rana, "A taxonomy of software composition mechanisms," in *Proc. 36th EUROMICRO Conf. Softw. Eng. Adv. Appl. (SEAA)*, Lille, France, Sep. 2010, pp. 102–110.

[27] K.-K. Lau, F. M. Taweel, and C. M. Tran, "The w model for component-based software development," in *Proc. 37th EUROMICRO Conf. Softw. Eng. Adv. Appl.*, Sep. 2011, pp. 47–50.

[28] K.-K. Lau, P. V. Elizondo, and Z. Wang, "Exogenous connectors for software components," in *Proc. 8th Int. SIGSOFT Symp. Compon.-Based Softw. Eng.*, 2005, pp. 90–106.

[29] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[30] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proc. 22nd Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2000, pp. 178–187.

[31] F. Nejati, A. A. abdul ghani, N. Yap, and A. Jaafar, "Putracom: A concurrent component model with exogenous connectors," *IEEE Access*, vol. 6, pp. 15446–15456, 2018.

[32] I. Ntalamagkas, "Software component model with concurrency," Ph.D. dissertation, School Comput. Sci., The Univ. Manchester, Manchester, U.K., 2009.

[33] *Web Services Business Process Execution Language Version 2.0*, OASIS, Noida, Uttar Pradesh, Apr. 2007.

[34] G. Papadopoulos and F. Arbab, "Coordination models and languages," CWI, Amsterdam, The Netherlands, Tech. Rep. 10.5555/869262, 1998.

[35] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[36] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating," in *Proc. 4th Int. Conf. Configurable Distrib. Syst.*, Washington, DC, USA, 1998, pp. 43–52.

[37] T. Rana, "Incremental construction component-based systems: A study based current component model," Ph.D. dissertation, School Comput. Sci., The Univ. Manchester, Manchester, U.K., 2015.

[38] T. Rana, "Ex-man component model for component-based software construction," *Arabian J. Sci. Eng.*, vol. 24, pp. 1–14, Oct. 2019.

[39] T. Rana, Y. A. Bangash, A. Baz, T. A. Rana, and M. A. Imran, "Incremental composition process for the construction of component-based management systems," *Sensors*, vol. 20, no. 5, p. 1351, Feb. 2020.

[40] T. Rana, Y. A. Bangash, and H. Abbas, "Flow constraint language for coordination by exogenous connectors," *IEEE Access*, vol. 7, pp. 138341–138352, 2019.

[41] T. Rana and A. Baz, "Incremental construction for scalable component-based systems," *Sensors*, vol. 20, no. 5, p. 1435, Mar. 2020.

[42] N. Russell, A. ter Hofstede, and W. van der Aalst, "NewYAWL: Specifying a workflow reference language using coloured Petri nets," in *Proc. 8th Workshop Tutorial Practical Use Coloured Petri Nets CPN Tools*, K. Jensen, Eds., Aarhus, Denmark, 2007, pp. 1–8.

[43] J. Sametinger, *Software Engineering With Reusable Components*. New York, NY, USA: Springer-Verlag, 1997.

[44] P. Stepan and K.-K. Lau, "Controller patterns for component-based reactive control software systems," in *Proc. 15th ACM SIGSOFT Symp. Compon. Based Softw. Eng.*, 2012, pp. 71–76.

[45] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York, NY, USA: Addison-Wesley, 2002.

[46] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, and J. E. Robbins, "A component- and message-based architectural style for GUI software," in *Proc. 17th Int. Conf. Softw. Eng.*, 1995, pp. 295–304.

[47] C. Tran, "Composition operators for components Web services," Ph.D. dissertation, School Comput. Sci., Univ. Manchester, Manchester, U.K., 2011.
[48] P. Velasco Elizondo and K.-K. Lau, "A catalogue of component connectors to support development with reuse," *J. Syst. Softw.*, vol. 83, no. 7, pp. 1165–1178, Jul. 2010.
[49] T. Wang, "A context-sensitive service composition framework for dependable service provision in cyber-physical systems," *Int. J. Ad Hoc Ubiquitous Comput.*, vol. 24, no. 4, p. 1, 2017.

TAUSEEF RANA received the B.Eng. and M.Sc. degrees from London South Bank University, and the Ph.D. degree from The University of Manchester. He has been working with the software development industry for two years. He is currently serving as an Assistant Professor with the Computer Software Engineering Department, MCS [a constituent college of the National University of Sciences and Technology (NUST)]. His research interests include software development, programming languages, and distributed systems.

ABDULLAH BAZ (Senior Member, IEEE) received the B.Sc. degree in electrical and computer engineering from Umm Al Qura University (UQU), in 2002, the M.Sc. degree in electrical and computer engineering from Kerala Agricultural University, in 2007, and the M.Sc. degree in communication and signal processing and the Ph.D. degree in computer system design from Newcastle University, in 2009 and 2014, respectively. He was a Vice-Dean and then the Dean of the Deanship of Scientific Research with UQU, from 2014 to 2020. He is currently an Assistant Professor with the Computer Engineering Department, a Vice-Dean of DFMEA, the General Director of the Decision Support Center, and the Consultant of the University Vice Chancellor with UQU. His research interests include VLSI design, EDA/CAD tools, coding and modulation schemes, image and vision computing, computer system and architecture, and digital signal processing. Since 2015, he has been serving as a Review Committee Member of the IEEE International Symposium on Circuits and Systems (ISCAS) and a member of the Technical Committee of the IEEE Very Large Scale Integration (VLSI) Systems and Applications. He served as a Reviewer in a number of journals, including the IEEE Internet of Things, *IET Computer Vision*, *Artificial Intelligence Review*, and *IET Circuits, Devices and Systems*.