DYNAMIC INTERACTION OF INFORMATION SYSTEMS
Weaving Architectural Connectors on Component Petri Nets

Nasreddine Aoumeur, Gunter Saake
ITI, FIN, Otto-von-Guericke-Universität Magdeburg, Postfach 4120, D–39016 Magdeburg, Germany
{*aoumeur—saake*}*@iti.cs.uni-magdeburg.de*

Kamel Barkaoui
Laboratoire CEDRIC, CNAM, 292 Saint Martin, 75003 Paris - FRANCE
barkaoui@cnam.fr

Keywords: High-Level Nets, rewriting logic, specification/validation, architectural techniques, dynamic evolution.

Abstract: Advances in networking over heterogenous infrastructures are boosting market globalization and thereby forcing most software-intensive information systems to be fully distributed, cooperating and evolving to stay competitive. The emerging composed behaviour in such interacting components evolve dynamically/rapidly and unpredictably as market laws and users/application requirements change on-the-fly both at the coarse- type and fine-grained instance levels.
Despite significant proposals for promoting interactions and adaptivity using mainly architectural techniques (e.g. components and connectors), rigorously specifying / validating / verifying and dynamically adapting complex communicating information systems both at type and instance levels still remains challenging. In this contribution, we present a component-based Petri nets governed by a true-concurrent rewriting-logic based semantics for specifying and validating interacting distributed information systems. For runtime adaptivity, we enhance this proposal with (ECA-business) rules Petri nets-driven behavioral connectors, and demonstrate how to dynamically weaving them on running components to reflect any emerging behavior.

1 INTRODUCTION

Advances in networking over heterogenous infrastructures are forcing most software-intensive systems to be fully distributed, cooperating and evolving in facing the fierce struggle to stay competitive. This situation is more acute for contemporary information systems where market globalization and volatility are pressing business laws, policies (makers) and users requirements to adapt/evolve on-the-fly, unpredictably and rapidly. These facts have been urging organizations to shift from their traditional integrated form with centralized control to *loosely-coupled* networked applications owned and managed by diverse business partners. *Dynamic* business links with different organizations have thus to be dynamically set up and managed to satisfy customers, with on-the-fly inter-organizational collaboration and eventually outsourcing of activities to external service providers.

With the huge difficulties of directly and satisfactory implementing such complex communicating systems even with the availability of advanced networked infrastructures and standards (e.g. Web-Services and Service-oriented computing), more research efforts are nowadays devoted to foundational early phases of specification, validation and verification. Additionally, the tedious challenges of adaptivity must be tackled at these requirement early phases; otherwise any initial certified specification becomes rapidly outdated and useless, where afterwards the standing-alone programmers will be on charge of uncontrolled/unwished/untractable maintenance.

Architectural techniques over component-orientation belong to the mostly investigated and appropriate software-engineering conceptual means to address the adaptivity through their first-class explicit connectors (Cheng and Garlan, 2001; Szyperski, 1998) and their reconfigurations. Nevertheless, due to the growing complexity and volatility of such complex interacting software-intensive systems, it seems we are still far from a satisfactory architectural-based proposal. Indeed, existing proposals either focus on the coarse-grained component level (adding/removing/replacing components) and ignoring the component instance level (changing functionalities and behavior) or vice-versa. As will be demonstrated in this paper both levels are vital for expressing dynamic change and evolution.

With respect to information systems, the authors are not aware of any approach that handles dynamic adaptivity by respecting cross-organizational business rules (Wan-Kadir and Loucopoulos, 2003) at the architectural level at both the type and instance levels. Business rules are the main driving force for reshaping inter-organizations goals, policies and functioning/regulations and are therefore rapidly evolving to enhance the competitiveness. Nevertheless, even with respect to cutting edge service-oriented architectures and (web-) information system services, business rules-driven proposals are starting to emerge as leading conceptual means to capture change and adaptivity at domain level (G.Meredith and S.Bjorg, 2003; Charfi and Mezini, 2004; Cibran and Verheecke, 2005; Rosenberg and Dustdar, 2005).

This paper puts forwards a (business-) rule-based architectural approach for specifying / validating and dynamically adapting complex gross-organizational information systems. Methodologically, given a (UML-based) semi-formal description of the applications, we first propose a formal specification based on a variant of component-based Petri nets, we introduced in (Aoumeur and Saake, 2002). In this so-called CO-NETS framework, IS components are conceived as a hierarchy of modular classes we explicit observed interfaces. Secondly, for validating such components, a true-concurrent rewriting logic-based is proposed for governing different transitions behavior. So, besides distributed graphical animations, formal deduction proofs are derived for consistency and verification purposes.

Besides these specification / validation and verification capabilities, we propose in this paper the concept of rule-driven architectural connector behavior. Such architectural connectors have to reflect different ubiquitous cross-organizational and thus inter-component domain business rules. As we separately and explicitly specify such connectors, we are able to dynamically weaving the right interconnections when required to reflect the enforcement of new/modified policies, laws and other requirements. Crucial to point out is that the proposed dynamic weaving is non-intrusive, that is, we do not delve inside component functionalities or change/adapt any existing component internal behavior. Only observed component properties using explicit interfaces are required.

The rest of the paper is organized as follows. In the second section we review the main CO-NETS features and illustrate them with a simplified banking system. The third section introduce the concept of (business) rule-based architectural connectors at a semi-formal level. The fourth main section focuses on the modeling of such connectors within CO-NETS and how they are dynamically woven on interacting components. This paper is closed by some remarks and insights about future extensions. We should note that the paper's presentation is kept at an informal level with some hints when required to the formal counterpart.

2 CO-NETS components : Overview with illustrations

First we recall some structural aspects of the CO-NETS approach by presenting how conceiving CO-NETS component signatures. These structural aspects are then extended by behavioral features leading to CO-NETS components. The construction of components as hierarchy of classes through simple inheritance is also sketched. The derivation of complex interacting CO-NETS components using inter-component cooperation is then addressed. In the last subsection, we give how corresponding rewriting rules governing each component behaviour or their interaction is derived.

2.1 CO-NETS component signatures

A component signature defines the structure of object states and operations to be accepted by such states. The CO-NETS signature that we proposing can be informally described as follows:
Object states are algebraic terms of the form:

$$\langle Id | l_1 : v_1, ..l_k : v_k, f_1(Id), .., f_l(Id), s_1 : v'_1, .., s_t : v'_t \rangle$$

- Id represents an observed object identity taking its values from an appropriate abstract data type we denote OId.
- $l_1, .., l_k$ are considered to be local (i.e. hidden from the outside) attribute identifiers with respective current values $v_1, .., v_k$.
- $f_1(Id), .., f_l(Id)$ are attribute identifiers as 'functions' representing hidden values which cannot be observed even at the component level itself. Such attributes are therefore defined in a co-algebraic way (Hensel et al., 1998).
- The observed part of an object state is identified by the set of attributes $s_1, ..., s_t$; their associated current values are denoted by $v'_1, .., v'_t$.
- All attribute identifiers (local or observed) are defined as appropriate subsorts of a generic sort denoted AId, and their respective values are ranged over by the sort $Value$ (with $OId < Value$ to allow object valued attributes).

- For exhibiting intra-object concurrency and separating at any time local attributes from observed ones, we introduce a simple deduction rule we call 'object-state splitting / merging' rule that permits to split (resp. recombine) the object state as required.

- We also make an explicit distinction between *internal* messages and *external* as imported / exported messages.

With respect to this perception each template signature is henceforth endowed with an explicit interface composed of observed attributes and messages, and that we subsequently refer to as (basic) *component* signature.

2.1.1 The account and customer component signatures

We assume having accounts and customers that we naturally regarded as two separate yet interacting components. For instance, for the account component signature, each (current) account is composed of: a balance (shortly `bal`) as observed, and a minimal limit (`lmt`), a boolean value (`Red`) valuated to true when the balance goes below the minimal limit as local, its hidden PIN as observed function, account holder observed identity (`Hd`), and a list of pairs '[money, date]' (`Hs`) for recording performed (debit or credit) operations on an account. As observed messages are the debit (`Deb`), credit (`Crd`) and transfer (`Trs`) of money between two accounts, while as local message we allow the minimal limit attribute to be changed using (`Chglm`).

In the same spirit the corresponding Customer component algebraic signature is detailed in the appendix.

```
obj Account-data is
  protecting Real+ Date Nat Bool .
  subsort Money < Real+ .
  subsort History < List-History.
  op [] :  → History .
  op [_,_] : Money Date → History .
  op _._ : History List-History → List-History.
endo.
```

```
obj account is
  extending object-state .
  protecting Account-data .
  sorts Acnt .
  subsort Id.Acnt < OId .
  subsort DEB CRD TRS < Obs_Msg.
  subsort ChgL < Loc_Msg.
  subsort loc_Acnt obs_Acnt < Acnt < object .
  (* attributes as functions*)
    op Pin : Id.Acnt → string .
  (* Local attributes *)
op ⟨_ | Bal : _, Lmt : _, Hs : _⟩ : Id.Acnt
       Money Money History → loc_Acnt.
  (* observed attributes *)
op ⟨_ | Hd : _⟩ : Id.Acnt OId → obs_Emp .
  (* Local messages *)
op Chgl : Id.Acnt Money → ChgLm .
  (* observed messages *)
op Deb : Id.Acnt Money Date → DEB.
op Crd : Id.Acnt Money Date → CRD.
op Trs : Id.Acnt Id.Acnt Money Date → TRS.
  vars B, L, W, D : Money ; C : Id.Acnt .
endo.⧫
```

The declared variables will be used in the corresponding Co-nets component specifications as will be detailed here after.

2.2 Co-nets component specification

On the basis of a given component signature, we define the notion of component specification as a Co-net in the following straightforward way.

- Co-nets places are precisely defined by associating with each message declaration or method one (message) place, that is, such messages places contain associated message instances sent or received by objects but not yet performed. Also, with each object sort a (object) place is associated, that is, an object place contains current object states with respective attribute values. We note that places corresponding to external messages are drawn with bold circles.
- Co-nets transitions reflect the effect of messages on object states they are addressed to. Conditions may be associated to them restricting their application. Moreover, we distinguish between local and external transitions. Local transitions reflect object states change in a given component, whereas external ones capture the interaction between different components.

For building components as a *hierarchy* of classes with explicit interfaces, our conceptualization of simple inheritance, for instance, may be sketched as follows:

- The signature of a given subclass is constructed in the same spirit as for (super-classes) Co-nets signatures, but regarding it as an extension or enrichment of the superclass signature.
- The corresponding net of the subclass is similarly constructed by associating with each message a corresponding (message-)place and an (object-)place for gathering the object-states composed of the proper attributes in this subclass. In the same way transitions are associated with these messages following their informal meaning, but with the possibility that superclass object-places may be participating as input or output arcs.

Other high-level Petri nets variants include (Reisig, 1991), (H. Ehrig and Ribeiro, 1994), (Jensen, 1992), (Biberstein et al., 1997), (Valk, 2001) (among others).

Application to the account and customer components. The associated Co-nets account component is depicted in the left-hand side of Figure 1. This component is composed of current accounts as a superclass and saving accounts as a subclass (where the interest could be increased through `Tinc` and money be moved from a saving account to a current account using `Tmvt`).

The right-side of this figure represents the customer component. As described above, in this net in addition to the object place ACNT that contains all account instances three message places namely ChgL, DEB and CRD have been conceived. The effect of each message is captured by an appropriate transition that takes into account just the relevant attributes. For instance, as reflected by the transition Tchg for changing the minimal limit of an account, the message $Chgl(Id, Nlm)$ enters into contact just with the Lmt attribute of this account, identified here by Id. Important to point out is that for different transitions (i.e. Tdeb, Tcrd, Ttrs), we have opted for very *elementary* (and stable) behaviour, so that complex behaviour could be derived rather as *contracts* between the holders and their accounts or between different accounts for a given account holder. For instance, the debit consists here just in decreasing the balance by the amount (i.e. without checking whether there is sufficient money, or the minimal limit is reached, or updating the history in consequence, etc).

2.3 Co-nets components interaction

By taking benefit of explicit interfaces (i.e. observed attributes and import / export messages) in each component, we present in this subsection how complex cooperative information systems may be composed of several truly distributed and independent yet *cooperating* Co-nets components.

As depicted in Figure 2, the general schema of 'external' transitions may be expressed as follows. Just relevant *external* parts of component states namely $\bigoplus_i \langle Id_{1_i} | bss_{1_i} \rangle, \ldots \bigoplus_j \langle Id_{p_j} | bss_{p_j} \rangle$ from components $Cp_1, ..., Cp_p$, enter into contacts with external (i.e. declared as imported or exported) messages to which they are sent, namely $ms_{i_1}, .., ms_{i_r}$ from such components. Under eventual conditions on attributes values and parameters messages, this result in the following: (1) the messages $ms_{i_1}, .., ms_{i_r}$ being consumed; (2) states of some external parts of objects participating in the communication being changed; and (3) new external messages (that may involve deletion/creation ones) being sent to objects at different components, namely $ms_{h_1}, .., ms_{h_r}$.

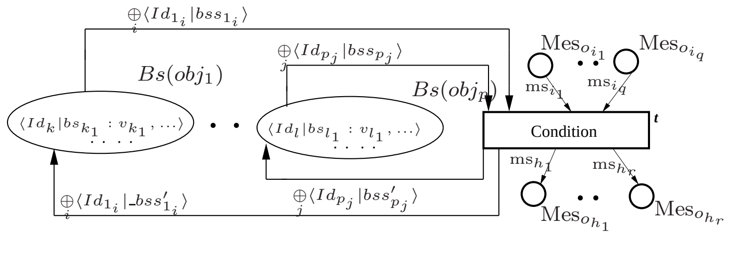

$\bigoplus_i \langle Id_{1_i} | bss_{1_i} \rangle$
$\bigoplus_j \langle Id_{p_j} | bss_{p_j} \rangle$
$Bs(obj_1)$
$Bs(obj_p)$
$\langle Id_k | bs_{k_1} : v_{k_1}, \ldots \rangle$
$\langle Id_l | bs_{l_1} : v_{l_1}, \ldots \rangle$
Condition
t
$\text{Mes}_{o_{i_1}}$
$\text{Mes}_{o_{i_q}}$
ms_{i_1}
ms_{i_q}
ms_{h_1}
ms_{h_r}
$\text{Mes}_{o_{h_1}}$
$\text{Mes}_{o_{h_r}}$
$\bigoplus_i \langle Id_{1_i} | \_bss'_{1_i} \rangle$
$\bigoplus_j \langle Id_{p_j} | bss'_{p_j} \rangle$

Figure 2: The Inter-component interaction pattern

Application to the running example. In Figure 3 the interaction of the two components is described using exclusively their observed features. We note here that besides those explicitly defined as observed features, attributes as functions are also observed, but their contents cannot be accessed either at local or at this interaction level. This concerns in particular the Pin attribute as function. Here are some comments on this observed interaction behaviour. The transition Twrd, for instance, captures the sending of a withdraw order from the customer, namely C-Deb(C,A,M), that have to be received by the withdraw message in the concerned account (i.e. the account of this customer as account hold). This means that we have to select from the observed account attribute through the read-arc the inscription:$\langle A | Hd : C \rangle$.

2.4 Animating and Validating Co-nets Specifications

One of the most advantage of the Co-nets approach is its operational semantics expressed in rewriting logic; moreover, by introducing the state splitting / recombining axiom there is a natural exhibition of intra-object concurrency. More precisely, each transition is governed by a corresponding rewrite rule interpreted in rewrite logic (Meseguer, 1992) (or more precisely an instantiation of this logic we refer to as Co-nets rewrite theory. The main ideas consist in: (1) associating with each marking mt its corresponding place p as a pair (p, mt); (2) introducing a new multiset generated by a union operator we denote by $\otimes$ for reflecting Co-nets states as multisets over different pairs (p_i, mt_i), , that is, a Co-nets state is described as $(p_1, mt_1) \otimes (p_1, mt_2) \otimes ...$; (3) allowing the distributivity of $\otimes$ over $\oplus$ for exhibiting a maximal of concurrency, that is, if mt_1 and mt_2 are two marking multisets then we always have: $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$; (4) enabling object states' splitting and recombining at a need for exhibiting intra-object concurrency.

Application to the account component. By applying the afore-described general form of rewrite rules, it is not difficult to generate the transition rules associated with the Co-nets specification of the banking example. The rewrite rules associated with the debit and credit messages (i.e. transitions Tdeb, Tcrd), for instance, take the form:

Tdeb :$(DEB, Deb(C, W)) \otimes (ACNT, \langle C | Bal : B \rangle) \Rightarrow (ACNT, \langle C | Bal : B - W \rangle)$

Tcrd :$(CRD, Crd(C, D)) \otimes (ACNT, \langle C | Bal : B \rangle) \Rightarrow (ACNT, \langle C | Bal : B + D \rangle)$

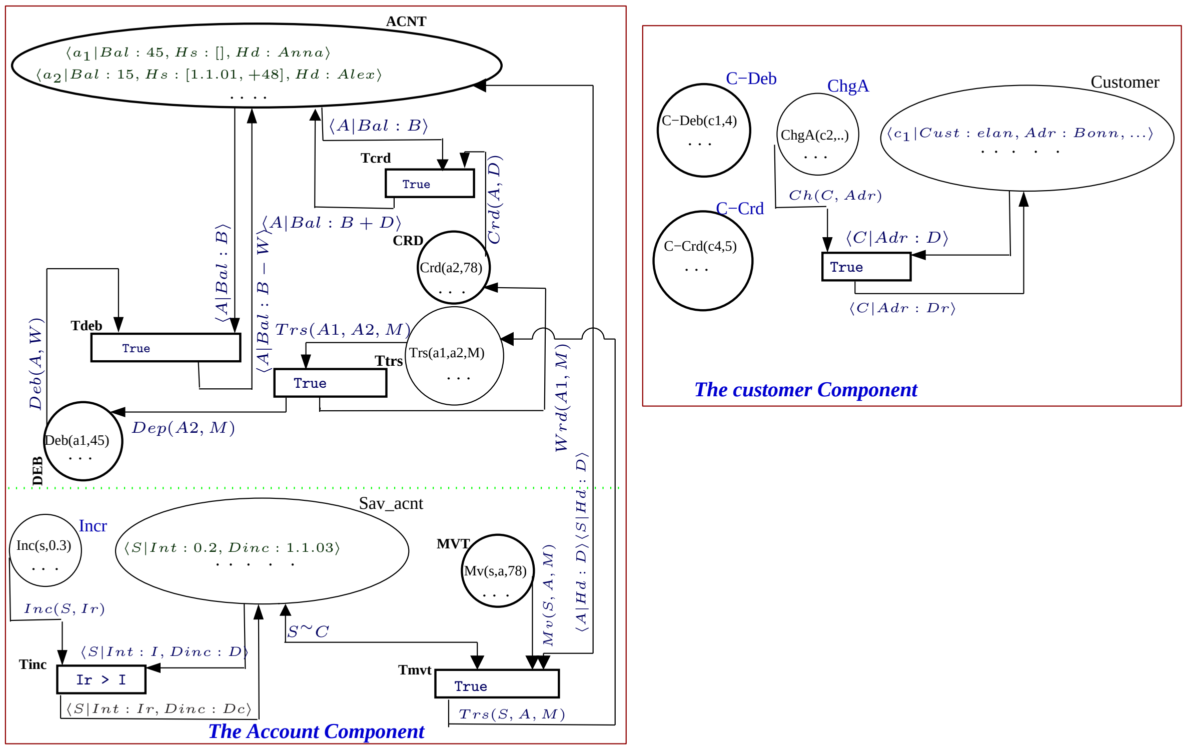


Figure 1: A Simplified Banking specification within Co-nets

3 ECA-rule based Architectural Connectors

Architectural connectors must generally endowed with: (1) *roles* expressing functional requirements from candidate components to interact; and (2) *glues* for expressing the behavior governing such inter-component interaction. As we are targeting complex information systems are main domain application, instead of just exchanging and ordering messages at this interaction level we propose a more knowledge-intensive behavioral interaction patterns expressed in terms of *business rules*. For such cross-organisations business rules, we propose the most adopted form which the Event-Conditions-Actions (ECA) paradigm. More precisely the general pattern, we decide to follow for expressing inter-component interactions is as follows:

ECA-behavioral glue <*glue-Identity*>
  interface participants <*list-of-participants*>
  invariant <*possible interaction constraints to respect*>
  constants/attributes/operations
    <*extra-required elements for the interaction*>
  interaction rules: <*Rule-Name*>
    at-trigger <(set-of-)events>
    under <conditions>
    reacting <set-of-actions>

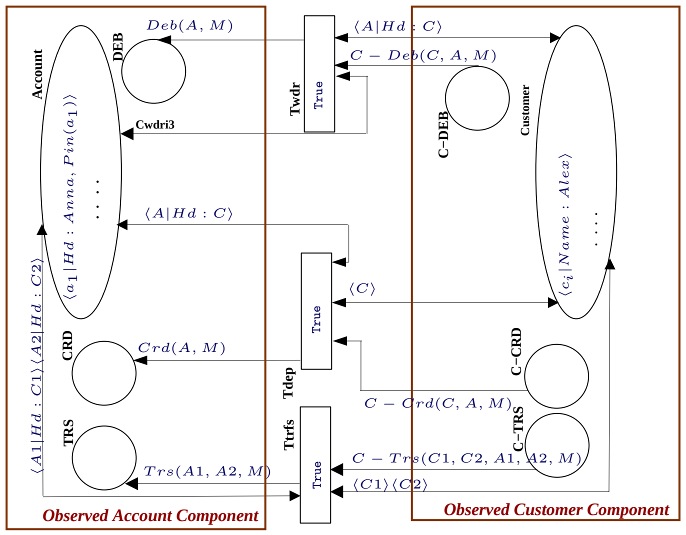
Important in this behavior-driven components interaction is above all the name of the entities participating in such coordination. Secondly, when required we have to specify invariants and constraints to be observed during the interaction. Thirdly, besides exchanged messages, statefull data and events between participants, to express complex interaction patterns we can define and use local to the glues properties such as constants, attributes and operations. For a given architectural connector glue, The ECA-based interaction rule itself starts by describing the event(s) triggering the interaction, then which conditions have to be fulfilled and finally what are the cooperative actions are to be performed.

This behavioral rule requires of course from different participating entities explicit interfaces including different events, messages and other properties (such as constants, variables ,etc). When needed, we explicitly specify such interfaces before giving this glue.

3.1 ECA behavioral glues : Illustration

To stay competitive banking systems are offering different incitive packages for their customers, ranging from simple agreed-on contracts (e.g. different formulas for withdrawing / transferring moneys) to highly sophisticated complex offers (i.e. staged housing loans, mortgages, etc.) depending on their profiles, trust, experiences, etc.

Let us illustrate simple cases of customer-bank agreements using our running example, through two customer-tailored variants of withdrawals. The usual case for any ordinary customer is to check what is called standard



Account
DEB
$Deb(A, M)$
$\langle a_1|Hd : Anna, Pin(a_1)\rangle$
Cwdri3
$\langle A|Hd : C\rangle$
CRD
$Crd(A, M)$
TRS
$Trs(A1, A2, M)$
$\langle A1|Hd : C1\rangle\langle A2|Hd : C2\rangle$
Observed Account Component
Twdr
True
Tdep
True
Ttrfs
True
$\langle A|Hd : C\rangle$
$C - Deb(C, A, M)$
C–DEB
Customer
$\langle c_i|Name : Alex\rangle$
$\langle C\rangle$
C–CRD
$C - Crd(C, A, M)$
C–TRS
$C - Trs(C1, C2, A1, A2, M)$
$\langle C1\rangle\langle C2\rangle$
Observed Customer Component

Figure 3: The Customer account Co-nets components interaction

withdrawal, where the customer has to be the owner of a corresponding account and the withdrawal amount should not go beyond the current balance. Adopting our behavioral glue notations, this agreement could expressed as follows. In this interaction, the participants are interfaces from the customer and the account component business entities. The account interface have to display the balance attribute as hidden function (no current amount is seen) and the operation debit. At the customer interface, we have to identify the withdrawal triggering event and the ownership invariant (these obvious interfaces as just skipped here).

```
ECA-behavioral glue Std-Withdraw
participants Acnt:  Account; Cust:  Customer
invariants Cust.own(Acnt) = True

interaction rule :  Standard
at-trigger Cust.withdraw(M)
   under  (Acnt.bal() ≥ M)
   acting Acnt.Debit(M)
end Std-withdraw
```

A more flexible withdrawal for moderately previleged customers is to endow them with a credit those amount depends on profile and trust. Customers enjoying such agreements can now withdraw amounts going beyond the current balance. The modelling of such flexible agreed-on withdrawal takes the following slightly modified behavioral glue.

```
ECA-behavioral glue VIP-Withdraw
participants Acnt:  Account; Cust:  Customer
attribute Cust.credit :  Money
invariants Cust.own(Acnt) = True

interaction rule :  VIP
at-trigger Cust.withdraw(M)
   under  (Acnt.bal() +Cust.credit ≥ M)
   acting Acnt.Debit(M)
end VIP-withdraw
```

4 Modelling ECA Behavioral Connectors as extended Co-nets

With Co-nets capabilities in capturing statefull components behavior, we present in the following how ECA-driven architectural connectors enhance these potentials towards more dynamic adaptivity and evolution. Following the same intuitive guidelines for constructing Co-nets components from informal component-based applications, the modelling steps for integrating such architectural connectors into already specified Co-nets components could sketched in the following. First, we have derive from a given ECA-based architectural connector description, a more precise corresponding component signature specification by algebraically specifying different properties (attributes, messages, events, etc.). Secondly, by gathering different connector attributes and participants into states, we then associate such each state type a corresponding place and with each messages and operations also a place. This results in the skeleton of the Petri net for such architectural connectors. Finally, we have to inject the rules into such skeleton by assigning conditions to transition conditions, events as input arc-inscriptions and actions as output ones. In a more detail, these translating steps could be explained as follows:

1. Define architectural connector structure algebraically using the Co-nets component signature pattern. That is, first, gather all component participants interface identities with possible other attributes into a glue state type-as-tuple.
2. Specify all involved messages, events, constants and

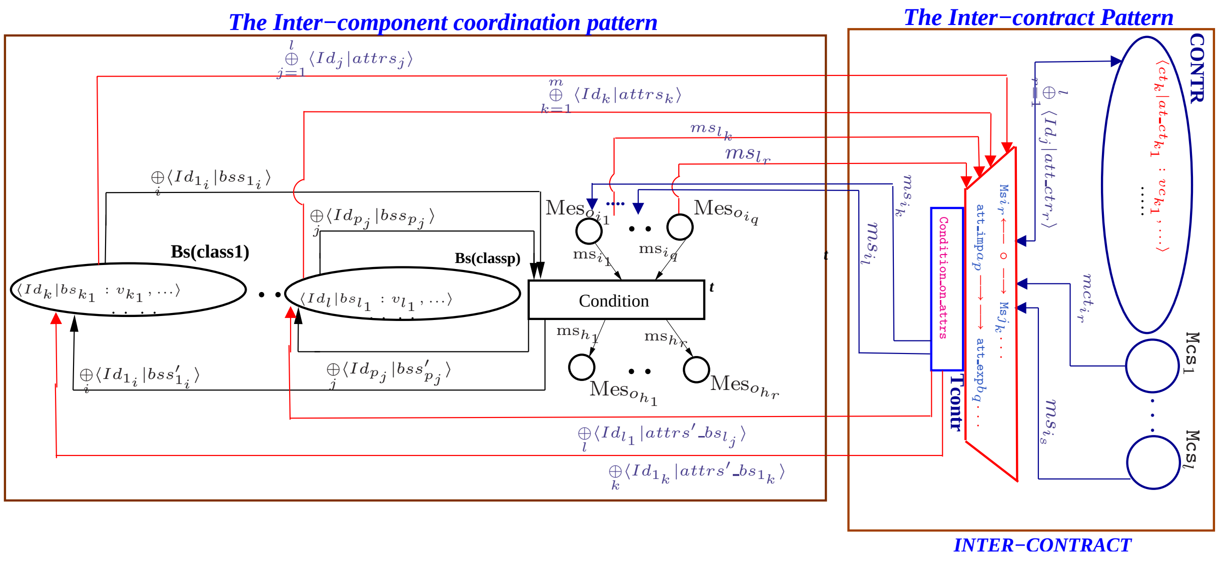


Figure 4: The general CO-NETS inter-contracts class pattern and inter-component transitions

invariants in a given architectural connector using a precise algebraic setting.

3. Associate with each state type a given "glue-state" place, and with each local messages and events a given place.

4. The transitions for architectural connectors have to be constructed for reflecting the different ECA-rules. The general pattern for such interacting transitions is to be conceived following now the new general pattern we depict in Figure 4. This pattern expresses the idea of dynamically weaving exogenous behavior of interacting components. Thats is:

   - Involved component interfaces in a given ECA architectural behavior are captured using observed state parts (places) from such CO-NETS components and imported/exported messages (places).
   - These interface elements (e.g. observed states and messages) enter into contact to reflect the ECA-rule in question.
   - To allow renaming and refinement while weaving such architectural behavior, we add in the interaction transition new boxes for eventual term assignments.
   - To semantically govern such assignments, we enhance transition rewrite rules using *matching conditions* as proposed for the computational logical language ELAN (van den Brand et al., 2002), that is interpreted by conditional rewriting in rewriting logic (Borovansky et al., 2002). Rules in rewriting logic with matching conditions take the following form

     $$[l] : l \longrightarrow r \quad \texttt{where} \quad p_1 := c_1 \ \ldots \ p_n := c_n$$

     Where in the assignment $p_i := c_i$, p_i and c_i are terms of the same sort. For rewriting a term t with such a rule first as usual a subterm $t|_p$ has to match l through a substitution σ_i, and then the matching $p_i := c_i$ has to be checked. This checking returns to compute $c_i\sigma_i$ (and then a normal form for it if any) and verify that there is a matching of $c_i\sigma_i$ with p_i (we denote by μ). This matching μ is then composed with σ_i for rewriting the term t at position p, that is, $t' = t[p \leftarrow \sigma_i\mu(r)]$

Taking this semantics into account, the rewrite rule we associate with each inter-component transitions general pattern depicted in Figure 4 takes the form:

Tcontr: $(Bs(class_1), \bigoplus_{j=1}^{l} \langle Id_j|bss_j\rangle) \otimes$
$(Bs(class_p), \bigoplus_{k=1}^{m} \langle Id_k|bss_j\rangle) \otimes (CONTR, \bigoplus_{r=1}^{l} \langle Id_k|cntr_r\rangle) \otimes$
$(Ms_{o_ik}, ms_{l_k}) \cdots \otimes (Ms_{o_ip}, ms_{l_p}) \otimes (Mct_{i_r}, mst_{i_r}) \cdots \otimes$
$(Mct_{i_s}, mst_{i_s}) \quad \Rightarrow \quad (Bs(class_1), \bigoplus_{j=1}^{l} \langle Id_j|bss'_j\rangle) \otimes$
$(Bs(class_p), \bigoplus_{k=1}^{m} \langle Id_k|bss'_j\rangle) \otimes (CONTR, \bigoplus_{r=1}^{l} \langle Id_k|cntr'_r\rangle) \otimes$
$(Ms_{o_r}, ms_{l_r}) \cdots \otimes (Ms_{o_k}, ms_{r_k})$
`if` *Condition*
`where` $\quad ms_{i_r} := ms_{j_k} \ \ldots \ att\_ip_p := att\_op_q.$

4.1 Illustration using the running example.

We approach the two already conceived architectural within the CO-NETS framework following the above steps. That is, as shown below first their algebraic signatures are derived—the ECA-rule themselves are skipped and replaced by just informal text as they are to be specified through the architectural Petri net afterwards.

```
obj Std-Withdrawal .
  extending object-state .
  using Id.Acnt Id.Cust .
  subsort StdGlue  < object .
  (* StdGlue state *)
    op ⟨_ | Acnt : _, Cust : _⟩ :  Id.StdGlue
      Id.Acnt  Id.Custm→ StdGlue.
  vars Z : Money ; H : Id.Cust .
  vars A : Id.Acnt, C: Id.Cust, Gl :  Id.StdGlue .
endContract.♦
```

```
obj  VIP-Withdraw .
  extending object-state .
  using Id.Acnt  Id.Cust .
  subsort VIPglue  < object .
  (* VIPGlue state *)
    op CST_VIP  :  → Nat
    op ⟨_ | Acnt : _, Cust : _, Crd(_) : _⟩ :  Id.VIPGlue
      Id.Acnt  Id.Custm  Real  → VIPGlue.
  vars Z : Money ; H : Id.Custm .
  vars A : Id.Acnt, C: Id.Cust, Ct :  Id.VIPGlue .
  eq CST_VIP = Nat_Value
endContract.♦
```

To have a case where the assignment is to be applied, we are adding a new very abstract architectural connector[1], which through appropriate assignments may play the role of either Standard or VIP withdrawal. We denote by `inhibitor` this architectural connector, which posses two attributes `big` and `small` and an operation to restrict the application of such connector through a dynamic assignments of these two attributes to involved component properties.

```
obj  inhibitor  .
  extending object-state .
  sort RESTR
  Subsort Id.Inhib < OId
  subsort Inhib < object .
  (* observed imported attributes)
    op ⟨_ | big : _, small : _⟩ :  Id.Inhib
      integer  integer → INHIB.
  (* observed messages)
    restrict :  Id.Inhib → RESTR
endobj.♦
```

By assigning the `balance` to the abstract variable `big`, the withdrawn amount `M` to the variable `small`, we can straightforwardly emulate the standard architectural connector as depicted in Figure 5. In this figure, the interaction is no more direct between the two customer and the account components, but instead dynamically regulated through architectural connectors. Depending on the specific agreements of the bank with its customers, a withdrawal could be performed differently (for our simple case either standard or VIP).

[1]With such abstract connector we can go beyond ECA business rules and IS and specify any distributed embedded systems.

5 CONCLUSIONS

For *reliably* developing complex concurrent and dynamically evolving information systems, we extended component-based Petri nets with ECA-compliant behavioral architectural connectors. We shown how to incrementally incorporate different state-full connectors those behaviors being extracted from externalized cross-organizational business rules. Both graphical animations and concurrent symbolic computations using soundly enriched rewriting logic are possible for validating the conceived evolving conceptual model.

This first step towards enhancing component-based formalisms with dynamic inter-component explicit and state-full interactions in true-concurrent distributed environments has to be further worked out for resulting in a complete methodology with adequate software tools supporting it. The enrichment of the integration with CO-NETS meta-reflection capabilities we proposed in (Aoumeur and Saake, 2004) could be very beneficial to achieve a reactive self-adaptivity over such evolving architectural connectors. Another promising direction is to enhance the analysis capabilities of this extended CO-NETS by adapting the technique proposed in (Aoumeur et al., 2000).

REFERENCES

Aoumeur, N., Barkaoui, K., and Saake, G. (2000). On the Benefits of Rewrite Logic as a Semantics for Algebraic Petri Nets in Computing Siphons and Traps. In *Proc. of the 10th International Conference on Computing and Information (ICCI '2000), Kuweit. to appear in LNCS, Springer*.

Aoumeur, N. and Saake, G. (2002). A Component-Based Petri Net Model for Specifying and Validating Cooperative Information Systems. *Data and Knowledge Engineering*, 42(2):143–187.

Aoumeur, N. and Saake, G. (2004). Dynamically Evolving Concurrent Information Systems :A Component-Based Petri Net Proposal. *Data and Knowledge Engineering*, 50(2):117–173.

Biberstein, O., Buchs, D., and Guelfi, N. (1997). CO-OPN/2: A Concurrent Object-Oriented Formalism. In *Proc. of Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems(FMOODS)*, pages 57–72. Chapman and Hall.

Borovansky, P., Kirchner, C., Kirchner, H., and Moreau, P. (2002). Elan from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185.

Charfi, A. and Mezini, M. (2004). Hybrid web service composition: Business processes meet business rules. In Aioello, M., Aoyama, M., Curbera, F., and Papazoglou, M., editors, *Proceedings 2nd International Conference on Service Oriented Computing (ICSOC04)*. ACM Press.

Cheng, S. and Garlan, D. (2001). Mapping architectural concepts to uml-rt. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*.

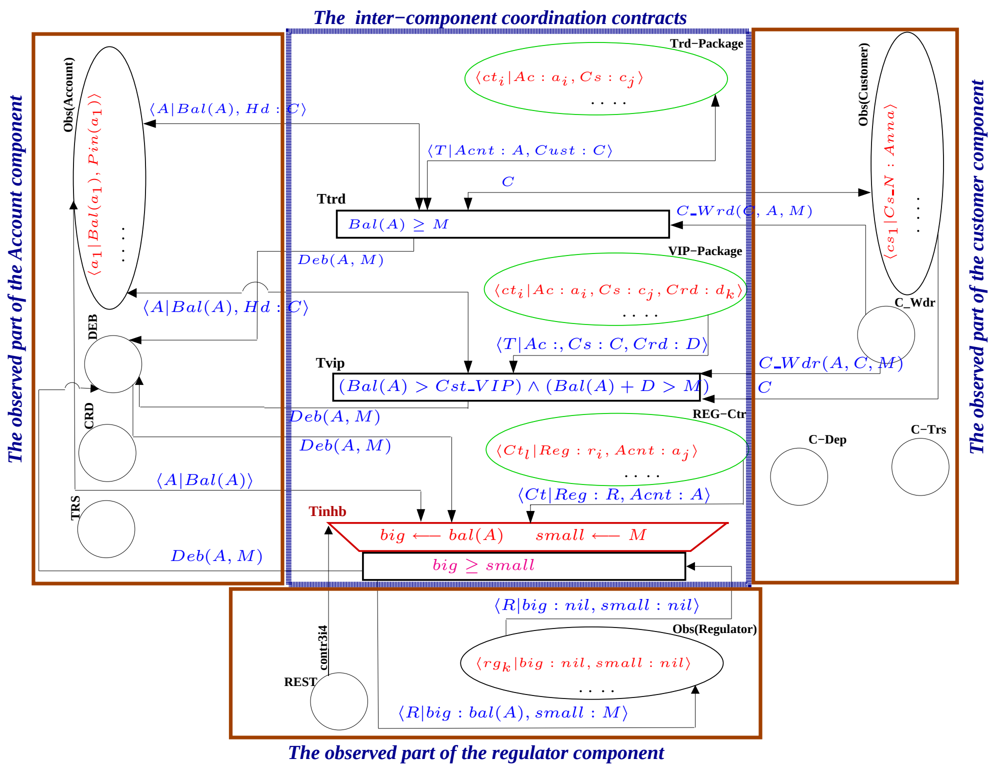

The inter−component coordination contracts
The observed part of the Account component
Obs(Account)
$\langle a_1|Bal(a_1), Pin(a_1)\rangle$
DEB
CRD
TRS
$\langle A|Bal(A), Hd : C\rangle$
$\langle A|Bal(A)\rangle$
$Deb(A, M)$
Trd−Package
$\langle ct_i|Ac : a_i, Cs : c_j\rangle$
$\langle T|Acnt : A, Cust : C\rangle$
Ttrd
$Bal(A) \geq M$
VIP−Package
$\langle ct_i|Ac : a_i, Cs : c_j, Crd : d_k\rangle$
$\langle T|Ac :, Cs : C, Crd : D\rangle$
Tvip
$(Bal(A) > Cst\_VIP) \wedge (Bal(A) + D > M)$
REG−Ctr
$\langle Ct_l|Reg : r_i, Acnt : a_j\rangle$
$\langle Ct|Reg : R, Acnt : A\rangle$
Tinhb
$big \longleftarrow bal(A) \quad small \longleftarrow M$
$big \geq small$
The observed part of the customer component
Obs(Customer)
$\langle cs_1|Cs\_N : Anna\rangle$
C_Wdr
C−Dep
C−Trs
C
$C\_Wrd(C, A, M)$
$C\_Wdr(A, C, M)$
$\langle R|big : nil, small : nil\rangle$
Obs(Regulator)
$\langle rg_k|big : nil, small : nil\rangle$
contr3i4
REST
$\langle R|big : bal(A), small : M\rangle$
The observed part of the regulator component

Figure 5: ECA-Architectural connectors woven on the Co-nets banking system

Cibran, M. and Verheecke, B. (2005). Dynamic Business Rules for Web Service Composition. In *Proc. of the Second Dynamic Aspects Workshop (DAW05), in join with AOSD05.*

G.Meredith and S.Bjorg (2003). Contracts and Types. In M.Papazoglou and (guest editors), D., editors, *Special Issue on Service-Oriented Computing, Communications of the ACM*, volume 46(10), pages 41–47.

H. Ehrig, J. P. and Ribeiro, L. (1994). Algebraic High-Level Nets - Petri Nets Revisited. In *Proc. Joint ADT-COMPASS Workshop*, volume 785 of *Lecture Notes in Computer Science*, pages 188–206. Springer.

Hensel, U., Huisman, M., Jacobs, B., and Tews, H. (1998). Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools. In Hankin, C., editor, *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 105–121. Springer.

Jensen, K. (1992). Coloured Petri Nets: Basic Concepts, Analysis Methods and practical Use - Volume 1 : Basic Concepts. *EATCS Monographs in Computer Science*, 26.

Meseguer, J. (1992). Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155.

Reisig, W. (1991). Petri Nets and Abstract Data Types. *Theoretical Computer Science*, 80:1–30.

Rosenberg, F. and Dustdar, S. (2005). Towards a Distributed Service-Oriented Business Rules System. In *Proc. of the of EEE European Conference on Web services (ECOWS)*. IEEE Computer Society Press.

Szyperski, C. (1998). *Component Software : Beyond Object-Oriented Programming*. Addision-Wesley.

Valk, R. (2001). Concurrency in communicating object petri nets. In Agha, G., de Cindio, F., and Rozenberg, G., editors, *Concurrent Object Oriented Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 164–165. Springer.

van den Brand, M. G. J., Moreau, P.-E., and Ringeissen, C. (2002). The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In van den Brand, M. G. J. and Lämmel, R., editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*. Electronic Notes in Theoretical Computer Science, Vol. 64, Nr. 3.

Wan-Kadir, W. and Loucopoulos, P. (2003). Relating Evolving Business Rules to Software Design. *Journal of Systems Architecture*.

APPENDIX

For the customer component, local attributes are the address, age and the monthly income (resp. `Adr`, `Age` and `M_incom`), and as observed attributes we consider the custumer name (`Name`). As a local message we have the changing of address, and as observed messages we assume that all account operations have to be initiated by the customer, namely `C-Deb`, `C-Crd` and `C-Trs`.

```
obj Customer is
  extending object-state .
  sorts Custom .
  subsort Id.Custm < OId .
  subsort C-DEB  C-CRD  C-TRS < Obs_Msg.
  subsort ChGA < Loc_Msg.
  subsort loc_Cust  obs_Cust < Custm < object .
  (* Local attributes *)
op ⟨_ | Adr : _, Age : _, M_Incom : _⟩ :  Id.Acnt
            Address  Age  Real→ loc_Cust.
  (* observed attributes *)
op ⟨_ | Name : _⟩ :  Id.Cust  String → Obs_Cust .
  (* Local messages *)
op ChgA : Id.Cust  Address → ChGA .
  (* observed messages *)
op C-Deb :  Id.Cust  Id.Acnt Money  Date → C-DEB.
op C-Crd :  Id.Cust  Id.Acnt Money  Date → C-CRD.
op C-Trs :  Id.Cust  Id.Acnt Money  Date → C-TRS.
  vars B, L, W, D : Money .
  vars C : Id.Acnt .
endo.⧫
```