Towards a Colored Reflective Petri-Net Approach to Model Self-Evolving Service-Oriented Architectures

Ying Liu
Northeastern University,
Liaoning, China
liuying@mail.neu.edu.cn

Walter Cazzola
DICo, Università di Milano,
Milano, Italy
cazzola@dico.unimi.it

Bin Zhang
Northeastern University,
Liaoning, China
zhangbin@mail.neu.edu.cn

ABSTRACT

Service-based software systems could require to evolve during their execution. To support this, we need to consider system evolving since the design phase. Reflective Petri nets separate the system from its evolution by describing it and how it can evolve. However, reflective Petri nets have some expressivity limits and render overcomplicated the consistency checking necessary during service evolution. In this paper, we extend the reflective Petri nets approach to overcome such limits and show that on a case study.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Coding Tools and Techniques—*Petri Nets*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Extensibility*

General Terms

Reflection, Software Evolution and Web Service

Keywords

Reflective Petri Nets, Service Composition, Adaptive SOA

1. INTRODUCTION

Since the introduction of the *web service* technology, building up a software system has turned into composing those services which implement application and business process in a distributed environment [1, 21]. Services exist on the Internet openly and independently, and cooperate with each other through composition. Each service exposes its interface to the other services but they could be different and incompatible and also how the client want to use the service can change from time to time. Moreover, a system composed by services must be aware of changes in its environment and adapt and evolve according to such changes and to functional or nonfunctional requirements. Such kind of a system is called *adaptive service-based software* (ASBS) system [18]. In its lifecycle, the system behavior evolves as soon as the changes to environment or customer requirements occur. How to analyze, design and validate the system and its behavior during the evolution is currently an open issue [12, 14].

As services are running in a dynamic and open environment, they should be able to dynamically evolve. Since how the system evolves during its execution is unpredictable, it is necessary to consider the evolution since the earlier stages of service development by considering it as part of service behavior. Mostly all formal methods like process algebra [9] and Petri nets [7] only model the system and its state without considering its evolution, when the system evolves, we have to re-model it. This breaks the relationship among the system before and after its evolution (no history available), and triggers problems in adapting the operations when evolving. To solve these problems, we can take into consideration the design of redundant paths for core parts [20] but this complicates the system and increases its running load. A more promising approach is to separate the evolution from the system itself and apply it when necessary. Reflection [10] technology is a mechanism to realize such a separation, and it is defined as the activity performed by an agent when doing computations about itself. Reflective Petri nets (RPN) [4, 5] is a Petri Nets based formalism that closely follows the characteristics of a reflective framework. RPNs can consider the requirement from system evolution at design-time and model the system and how it can evolve. RPNs satisfy the modeling needs of an ASBS system from the point of view of dynamic evolution.

When RPN simulates the evolutionary process of an ASBS system, we can check its consistency [13] by simulation models to ensure its correct running for on-line evolution. In RPN, the first layer is represented by a traditional place/transition Petri net (PN) modeling the software system prone to be evolved. The controlling construction of the software system is described by traditional PN flexibly, so we could check its controlling consistency by validating weak termination, appropriate termination, and dead activities of nets. Unfortunately RPN is too limited to easily deal with the complexity of the message structure and influences its data stream consistency. For example, the evolution of service composition could imply the deletion of service operations then we should check whether there are services that take the output of the removed operations as input but tokens cannot express the structure and content of a message, so it is very hard to check its data stream consistency. To tackle this issue we have to adopt a more expressive formalism to model service interface.

Compared with traditional PN, colored PNs [8] seem the perfect choice: more expressive than the place/transition PNs by adding colors and still a PN-based formalism that will limit the changes to the overall reflective PN model, and applied in modeling service composition system widely [15, 16, 19]. In fact the neces-

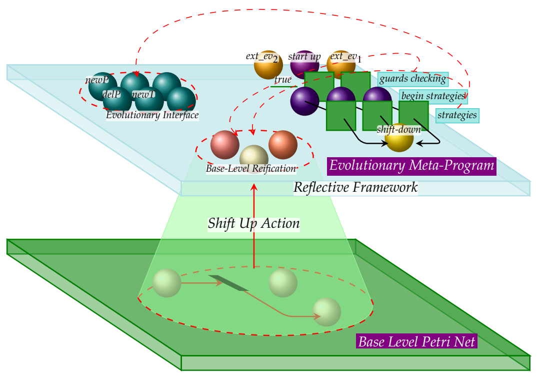

Figure 1: The architecture of the Colored reflective system.

sary changes should only affect how the reification is built and how the meta-level interacts with the base-level. This paper takes colored PNs as the base-level model, describes changed reification and interactions between base and meta levels, and proposes colored reflective Petri nets (CRPN) model. We use CRPN to model an ASBS system by giving concrete examples, and check controlling and data stream consistency for the system based on CRPN.

The rest of the paper is organized as follows. In Sect. 2 we introduce the reflective Petri net model to get the reader acquainted with the used terminology. In Sect. 3 we show our case study and highlight the limits of the reflective Petri nets. In Sect. 4 we show how the model is extended and apply the novel model to the case study. In Sect. 5 we give the consistency checking method by applying the novel model and proof of proposed theorems. In Sect. 6 we compare some related works with ours. Finally in Sect. 7 we draw our conclusions.

2. BACKGROUND

Reflective Petri-nets are structured into two logical layers. In the first layer, called *base-level*, runs a PN which models the system prone to be evolved, also called base-level PN; whereas in the second layer, *meta-level* runs the evolutionary meta-program according to the reflection mechanism. The reflective framework reifies the base-level PN into the meta-level as marking of a subset of places, called *base-level reification*. The basic operations on the base-level reification are part of the *evolutionary interface*, and a valid sequence of calls to them is called *evolutionary strategy*; several strategies compose the meta-program.

The evolutionary strategies in the meta-level drive the evolution of the base-level PN when certain events occur. Entities on the meta-level perform computations on entities residing on the lower level. The reflective framework, realized by a PN as well, is responsible for really carrying out the evolution of the base-level PN at the meta-level. Meta-level computations in fact operate on a representative of the lower-level, called reification. The base-level PN reification is defined as a marking of the reflective framework, and is automatically updated every time the base-level Petri net enters a new state. The reification is used by the meta-program (in the specific by the evolutionary strategies) to observe (*introspection*) and manipulate (*intercession*) the base-level PN.

Each change to the reification is reflected on the base-level PN at the end of a meta-computation (*shift-down action*), i.e., the base-level PN and its reification are causally connected, the reflective framework being responsible for that. According to the reflective paradigm, the base-level PN runs irrespective of the meta-program, being not aware of the existence of a meta-level. The meta-program is implicitly activated (*shift-up action*), and a suitable strategy is then put into action, under two conditions: i) either when the base-level PN model reaches a given configuration, or ii) when triggered by an external/unpredictable event. The occurrence of such an event is modeled by putting a token in a given place.

Intercession on the base-level PN is carried out in terms of a minimal set of basic operations (the evolutionary interface), that permits any kind of base-level evolution to be emulated, both at structure (topology) and marking (current state) level: the meta-programmer can add/remove places, transitions and arcs, and freely move tokens all over the base-level PN places. The evolutionary strategy specifies arbitrarily complex transformation patterns for the base-level Petri net. The evolutionary strategies can be designed as Petri nets or by using an CSP-like formalism. Evolutionary strategies have a transactional semantics: either they succeed, or leave the base-level PN unchanged.

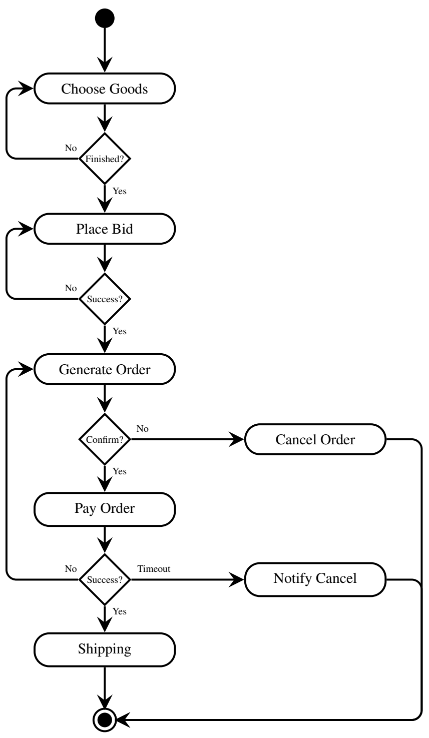

Figure 2: The workflow of the on-line shopping system.

3. CASE STUDY: ONLINE SHOPPING.

Let us consider an Online Shopping System composed of *customer management*, *goods management*, *order management* and *shipping management*. The main business workflow, realized by composing these services, is shown in Fig. 2. Following this workflow, a customer searches goods on-line, and chooses one to add into his shopping cart. Once goods have been chosen, he goes to place the bid, pays for them, chooses how to get them and fills the shipping address in. Once finished, the corresponding order will be generated and sent to the on-line shopping site. The clerk at the shop gets the ordered goods and arranges with a shipping company to send the goods according to the customer requirements.

The described workflow is intentionally naïve and does not support the shopper in checking if the goods available in the warehouse are enough to cover the order (*inventory management*) and when they could be shipped. To supports this feature, the system needs to evolve by composing with an *ad hoc* web service. In our example, to support the inventory management service, we have to adapt the existing on-line shopping system, by introducing the service before the shipping service in the main workflow.

The new service *inventory management* should check the quantities of the goods available in the warehouse and if they are sufficient to satisfy the client's order, the workflow continues to `shipping`; otherwise, it should freeze the order until enough goods are available in the warehouse and only then should unfreeze the order and deliver the goods.

Let us show how the RPNs can be used to model, check its control and data stream consistency, and then adapt the on-line shopping system. In this way we could safely evolve the workflow and only when we are sure to succeed updating the real system. As done in [7], the on-line shopping system is mapped into a PN model as follows:

1. The system workflow is represented as a P/T Petri Net.
2. Tokens model the messages exchanged among the services.
3. Transitions model the operations. An operation takes its input messages and produce its output messages through places called *in-place*s and *out-place*s respectively.
4. The connections between transitions, internal places and arcs are used to deal with the control flow.

Fig. 3(a) shows the PN model for the workflow in Fig. 2, and the PN in Fig 3(b) models the inventory management service. In-place and out-place of operations are represented as ai_j and ao_j respectively while the internal places are denoted as ap_j.

According to the definition of RPN, we can use an *evolutionary strategy* to compose services together and solve possible interface mismatches. The strategy, firstly, should add two operations `FreezeOrder` and `UnfreezeOrder` to the on-line shopping service then it will connect the services involved in the composition (input and output interfaces of `CheckInventory`, output interface of `PayOrder`, `AnnounceFreeze`, `AnnounceUnfreeze` and the input interface of `Shipping`, `FreezeOrder`, `UnfreezeOrder`). It verifies the compliance of the input interface of `CheckInventory` with the output interface of `PayOrder` and the compliance of the output interface of `DecreaseInventory` with the input interface of `Shipping`. If we compose these two services together, their interaction can be modeled as follows.

The on-line shopping service sends out the `Order` message; the inventory service receives the message and checks the inventory according to the goods id listed in the order:

– if the goods are available, it decreases the amount of the goods in the warehouse and the flow passes to the on-line shopping service that delivers the goods (`shipping`);
– otherwise it would send the `AnnounceFreeze` message but no operation in the on-line shopping service can receive and process it; consequently, the inventory service cannot receive any feedback and the flow cannot run correctly.

When composing two web services together as shown in Fig. 3, we have to check its control and data stream consistency for its correct running after the composition. The connection between two services satisfies control consistency when the resulting PN passes the reachability test, i.e., no dead nodes and no deadlocks are present. Instead, to check the data stream consistency we have to consider:

1. If the new service adds new operations (e.g., `AnnounceFreeze`) are their output correctly consumed by the old service?
2. Are the data used in the old system integrated with the new service? For example, could `Order` be marked as frozen?
3. Is there any interface mismatch between the connections of the services prone to be composed?

To consider the above questions, we have to analyze the semantics of the composite service, but traditional PNs offer a quite limited expressive power to describe such interfaces and the data the services share; in particular we cannot get detailed information about the type and value of the shared data. A (quite cumbersome) approach could be to code all the possible values on the number of tokens necessary to fire the corresponding transition. Although this solution is feasible it would increase the number of places and as a consequence the consistency checking will need more space and will become more complex. A more reasonable approach would consist in upgrading the formalism used to model the base-level from P/T PNs to colored Petri nets [8].

4. COLORED REFLECTIVE PETRI-NETS.

As showed in the previous section, modeling a composite service with a P/T PN introduces an undue complexity (a larger number of places and transitions, a scarce readability and tractability of the model, limited semantics checks, ...) to deal with, and, as a consequence, this hinders also its evolution (to compose a system with a new service implies to analyze and manipulate an excessively large PN and to sacrifice some checks). A more abstract representation of the exchanged messages will reduce the complexity of the model and of the corresponding evolutionary strategy. Colored Petri nets [8] extend Petri nets with the primitives for the definition of the data types (color) and the manipulations of data values.

Figure 4 shows the evolutionary framework from [4, 5] modified to deal with base-level systems modeled by colored Petri nets. The model performs a sort of concurrent-rewriting on the base-level, which is reified as a marking of the evolutionary framework. The places having prefix `BLreif` belong to the base-level reification, whereas those having prefix `EvInt` belong to the evolutionary interface. The basic color classes and initial marking need to be instantiated to establish a casual link between the evolutionary framework, the base- and the meta-level.

4.1 Color Definition.

Reification is an essential capability of all the reflective models. When the base-level system enters into a new state, the base-level reification is updated accordingly. The evolutionary meta-program uses the base-level reification to observe and manipulate the base-level system. Each change to the reification will be reflected on the base-level system at the end of a meta-program iteration. The meta-level maintains a set of data structures reifying the base-level computation. In colored reflective Petri nets, the reification is represented by a colored marking on a group of special places representing the places, transitions, marking and colors of the colored PN in the base-level.

Compared with PN, colored Petri nets have tokens with an algebraic structure. On one hand, this permits to simplify the base-level PN when the passed information has a relevance; on the other hand the reification of the marking in the meta-level must be adapted to deal with such an algebraic structure. As a consequence the reflective operations must be adapted as well. In the next, we will show how the reification and the operations on it (the *evolutionary interface*) have been extended to support colored PN in the base-level.

Letting $\mathcal{BL}$: (Σ_b, P_b, T_b, F_b, C_b, G_b, E_b, H_b, Π_b, M_0^b) be the base-level colored PN according to [8]. The basic color classes

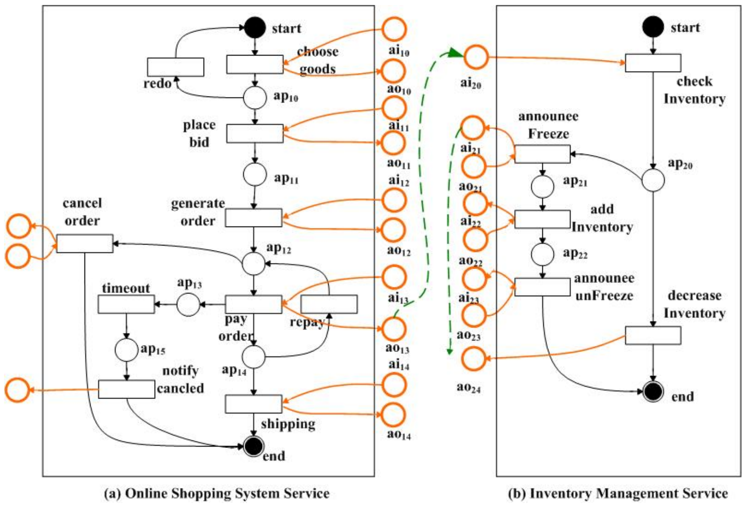

Figure 3: Services composition with a case of interface mismatch.

used in the reification are:

$$\begin{aligned}
\mathcal{C}_1 : NODE &= \underbrace{Place}_{P_b \cup UnNamedP} \cup \underbrace{Trans}_{T_b \cup UnNamedT} \cup \{Null\}, \\
\mathcal{C}_2 : ArcType &= i/o \cup h, \\
\mathcal{C}_3 : NodeColor &= C_b(Place) \cup G_b(Trans) \cup E_b(Trans).
\end{aligned}$$

The class *NODE* is partitioned in three groups: *Place*, *Trans* and *Null*. *Place* is P_b, i.e., the set of the base-level places, and *UnNamedP* that contains the places that might be added to the base-level without being explicitly named. The structure of *Trans* is analogously defined. The class *ArcType* defines two types of arcs, input/output and inhibitor. The definition of these two classes does not differ from the one in [4, 5]. Instead, the class *NodeColor* defines the color set function of *Place* and *UnNamedP* and related functions on color set. In the meta-level, the color domains for the base-level colored PN reification are:

$$\mathcal{C}_m(BLreif|Nodes) : NODE \cup NodeColor$$

$$\mathcal{C}_m(BLreif|Arcs) : \underbrace{NODE \times NODE}_{ARC} \times ArcType \times NodeColor.$$

The color for the reified nodes defines the type and color the node has in the base-level; the color for the reified arcs defines the type of nodes it connects in the base-level and its expression function.

4.2 Base-level Reification

The base-level reification takes place at system start-up and it is updated after the firing of each base-level transition. The base-level reification is defined as follows.

DEFINITION 1. *The reification* $reif(\mathcal{BL})$ *of a base-level Colored PN* $\mathcal{BL}$ *is the marking:*

$$\begin{aligned}
\mathcal{M}_m(BLreif|Nodes) &= \textstyle\sum_{n \in \mathrm{P}_b \cup \mathrm{T}_b} 1 \cdot n + 1 \\
\mathcal{M}_m(BLreif|Prio) &= \textstyle\sum_{t \in \mathrm{T}_b} (\Pi_b(\mathrm{t}) + 1) \cdot t \\
\mathcal{M}_m(BLreif|Marking) &= \sum_{p \in P_b} M_0^b(\mathcal{C}(p)_{MS}) + \sum_{t \in T_b} (E_b(t) + G_b(t))
\end{aligned}$$

$$\begin{aligned}
&\forall p \in \mathrm{P}_b, t \in \mathrm{T}_b : \\
&\mathcal{M}(BLreif|Arcs)(< p, t, i/o, C_b(p) >) = E_b(t) \\
&\mathcal{M}(BLreif|Arcs)(< t, p, i/o, G_b(t) >) = C_b(p), \text{iff}\, G_b(t) = True \\
&\mathcal{M}(BLreif|Arcs)(< p, t, h, \varepsilon >) = \mathrm{H}_b(p, t) \\
&\mathcal{M}(BLreif|Arcs)(< t, p, h, \varepsilon >) = 0
\end{aligned}$$

The places `BLreif|Nodes`, `BLreif|Arcs` and `BLreif|Prio` in definition 1 represent the base-level topology:

– the place `BLreif|Nodes` contains the base-level nodes and their colors;
– the tokens in `BLreif|Arcs` encode the connection between places and transitions in the base-level; e.g., the term <p_3, t_1, *i/o*, $C_b(p_3)$> denotes an input arc from place p_3 to transition t_1, with the marking $C_b(p_3)$;
– the marking of `BLreif|Prio` encodes the transition priorities.

If an evolutionary strategy invokes a shift down operation, the change operated on the markings will be reflected from the meta-level to the base-level. The marking of place `BLreif|Marking` defines the current state of the base-level, we use the color function of places, the expression function of arcs and guard function on transitions to denote it. The value of `BLreif|Marking` is initialized to the base-level initial state.

Name	Description
`EvInt|newP`	*Adding a (set of) place(s)*
`EvInt|delP`	*Removing a (set of) place(s)*
`EvInt|newT`	*Adding a (set of) transition(s)*
`EvInt|delT`	*Removing a (set of) transition(s)*
`EvInt|newA`	*Adding a (set of) arc(s)*
`EvInt|delA`	*Removing a (set of) arc(s)*
`EvInt|newC`	*Adding a (set of) colored token(s)*
`EvInt|delC`	*Removing a (set of) colored token(s)*

Table 1: Evolutionary interface of colored reflective PNs.

4.3 Evolutionary Interface.

The behavior associated to the evolutionary framework is very intuitive. Each operation defined by the evolutionary framework is represented by a place in the evolutionary interface used by the

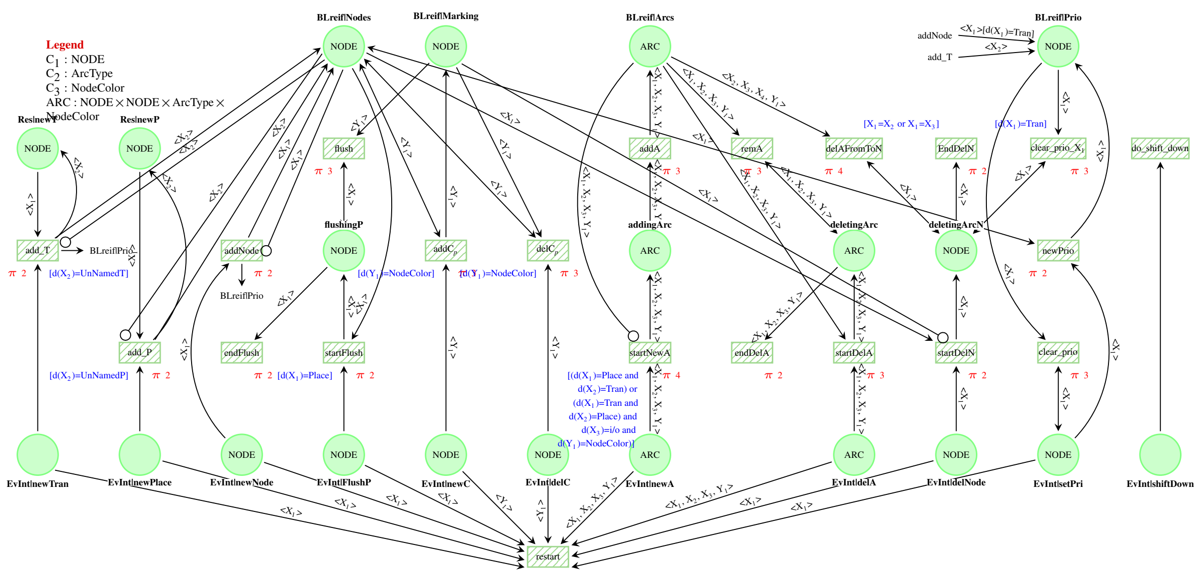
Figure 4: A view of the (new) evolutionary framework

strategies to activate the operation. Such a place is labeled by the `EvInt` prefix; when a token is pushed in one of these places a sequence of immediate transitions is triggered putting into action the corresponding command and changing the base-level reification as a result. There are eight basic actions summarized in Table 1. The first six actions come unmodified from [4, 5]; the remaining two operations are related to the colors. As an example, a token $<p_2>$ with color $\mathcal{C}(p_2)$ in place `EvInt|newC` corresponds to the command «add the token $<p_2>$ of color $\mathcal{C}(p_2)$ to the place».

The reflective framework carries out such actions in a consistent and atomic way, and they may have side effects. Let us consider for instance the addition of a new place, i.e., a token p_1 is put in the place `EvInt|newP`. The reflective framework, first, checks if the transition `addP` complete with true, then its color is added (transition `addC`), at last all the surrounding arcs and their colors and priorities are added (transition `addA`, `addC` and `newPrio`). If the given command is consistent with the current base-level configuration, it is carried out otherwise aborted and the model is restarted. The restart is implemented by firing a special transitions ($\mathtt{restart}_i$) and makes the whole meta-model go back to its state before last action. Any changes to the base-level topology can cause a restart operation, such as removing a non existing place/transition, add an already existing place/transition, adding a disconnected arc, and changes the color of nodes to a non existent color and so on.

4.4 Evolutionary Strategy.

When triggered by an external event, and/or when the base-level reaches a given configuration, a suitable evolutionary strategy specifying a set of transformations on the base-level can be put in action. To provide a convenient way to write a strategy, the framework provides a minimal CSP-like language. In this way, strategy designers would be unaware of the details of PN formalism, and the strategy could be automatically translated into a PN model.

5. CONSISTENCY CHECKING.

The extended reflective Petri net model supporting colored PNs in the base level are used to model our case study (the on-line shopping system) presented in Sect. 3. As explained two consistency aspects are of interest: the consistency of the control structure and the consistency of the data stream. The check on the control structure can be realized by analyzing weak termination, appropriate termination, and dead activities of nets, and by producing a symbolic reachability graph. Unfortunately due to lack of space we will not show this in details but we focus only on the validation of the data stream.

5.1 Data Stream Consistency Checking.

The data stream consistency checking needs to take the internal business rules, business process logic and constraints into consideration. The problems in data stream consistency checking includes four aspects.

- *Redundant data*: when a transition t produces a message d that is not consumed by the successive transition;
- *Data loss*: when both both transition t and t' produce the message d and t and t' are in different positions of the same concurrent structure in the process;
- *Missing data*: when the transition t requires a message d but none of the previous transitions produce such a message;
- *Data mismatch*: when two messages d and d' are semantically but not structurally equivalent.

According to the reflective characteristics of CRPN, the evolutionary strategy is carried out on the system reification in the meta-level, and then reflected on the base-level once the changes can be considered safe. In our case, the rules verifying the data stream consistency are:

- Rules for checking data redundancy:
  $\exists t \wedge t\bullet = ao_i$, in the process data are redundant if and only if
  $$C_b(ao_i) \in \mathcal{C}_3 \wedge C_b(ao_i) \nsubseteq \cup C_b(\bullet t_j) \wedge t_j \in post(t)$$
  is true;
- Rules for checking data loss:
  $$\exists p_i, p_j \in \mathcal{C}_1 \wedge \bullet p_i = \bullet p_j$$
  $$\exists p_l, p_m \in \mathcal{C}_1 \wedge p_l\bullet = p_m\bullet, p_l \in post(p_i), p_m \in post(p_j),$$
  in the process there is a data loss if and only if
  $$(d \in C_b(t_q) \wedge t_q \in post(p_i) \wedge t_q \in pre(p_l)) \wedge$$

$$(d \in C_b(t_r) \wedge t_r \in post(p_j) \wedge t_r \in pre(p_m))$$

is true;

– Rules for checking missing data:
$\exists t$, in the process data are missing if and only if

$$(d \in C_b(\bullet t)) \wedge (d \nsubseteq (C_b(\bullet n)) \cup C_b(n\bullet)) \wedge n \in pre(t))$$

is true.

In the case of a service composition system, the evolution may concern addition, cancellation and updating of a service. Since these operations may cause data stream inconsistency, we have to define some rules to validate the evolution against inconsistencies.

Rules about adding a service:

1. if there is a data loss in the service we are composing, we need to add some operations that produce the data necessary to the new service;
2. if there is data redundancy in the service we are composing, we need to add some operations that consume redundant data;
3. if there are missing data in the service we are composing, we need to change how the newly added service is composed.

THEOREM 1. *Let be Ω the model for an ASBS system written in CRPN that satisfies the constraints about data stream consistency. Let be ℓ an evolutionary strategies defined according to the given rules about adding a service. If ℓ is running on Ω and produce Ω' then Ω' still satisfies the constraints about data stream consistency.*

PROOF. If Ω' still satisfies the constraint of data stream consistency this means that there is data loss, missing data or redundant data in Ω'. Let us proceed by way of contradiction and suppose that there is a transition t' in Ω', and t' does not satisfy the constraints about data stream consistency.

– If the transition t' has a data loss issue, this means that t' requires a message d but d is never produced by a pre-transitions of t'; this violates the first rule about adding a service which requires that a new transition that produces d is added among the pre-transitions of t';
– If the transition t' has a redundant data problem, this means that t' produces a message d that is never consumed by a post-transitions of t' but this violates the second rule about adding a service, which requires to add a new transition among the post-transitions of t' that consumes d;
– If transition t' has a missing data problem, this means that the message d is produced by t' and another transition t'', and t' and t'' are in different branches of the same concurrent structure; but this violates the third rule about adding a service which requires to adjust the positions of t' and t'' in the process.

All the above cases are in contradiction with the hypothesis. Therefore, if ℓ is running on Ω and produce Ω', Ω' still satisfies the constraint of data stream consistency. $\langle\ \rangle$

Rules about deleting a service:

1. If deleting the transition t causes a data loss problem, we need to delete all the transitions involved in the data loss;
2. If deleting the transition t causes a missing data problem, we need to delete all the transitions involved in the problem.

THEOREM 2. *Let be Ω the model for an ASBS system written in CRPN that satisfies the constraints about data stream consistency. Let be ℓ an evolutionary strategies defined according to the given rules about deleting a service. If ℓ is running on Ω and produce Ω' then Ω' still satisfies the constraints about data stream consistency.*

Name	Description
`Adapter|recM`	*Receiving a message from the service*
`Adapter|storeM`	*Storing a message*
`Adapter|transM`	*Transforming a data message*
`Adapter|invokeS`	*Invoking a service*
`Adapter|sendM`	*Sending a reply message to the service*

Table 2: Basic actions in adapter

The proof of Theorem 2 is similar to the one for Theorem 1, due to lack of space the proof is omitted.

As the data mismatch problem makes the composition of two services infeasible, interfaces of composed services must be made compatible by building an adapter between them. To simplify the adapter construction, Benatallah *et al.* [2] put forward a mismatch pattern to capture and formalize the incompatibility. Patterns help us in identifying the differences and develop adapters based on the template of adaptation logic that resolves the captured mismatch. Every adapter has five basic actions, Table 2 describes such actions. In case of different mismatch patterns, different transformation rules for `Adapter|transM` are defined.

The evolutionary strategy for service composition is composed of the following steps:

1. check the compatibility of the interfaces; if they are compatible the service can be directly composed;
2. find to which mismatch pattern it belongs if the interfaces are incompatible and build up an adapter for the related pattern;
3. compose the two services through the just created adapter.

The mismatch patterns in [2] are characterized at the service protocol level from the structural perspective, so the adapter is also created from the structural perspective by a number of transformation definitions. Table 3 shows the six operators for different mismatch patterns that can be used in definition of the adapter.

In our case study (Sect. 3), we check its consistency against the four described problems and define the evolutionary strategy by the following CSP program.

```
pattern={pay order, shipping};
isolate(pattern);
freezeorder = newTrans();
unfreezeorder = newTrans();
ap_{16} = newPlace();
ai_{15}, ao_{15} = newPlace();
ai_{16}, ao_{16} = newPlace();
t_1, t_2, t_3, t_4 = newTrans();
addArc({<ap_{14}, freeorder, o>});
addArc({<freeorder, ap_{16}, i>});
addArc({<ap_{16}, unfreeorder, o>});
addArc({<unfreeorder, ap_{14}, i>});
addArc({<ao_{13}, recM, i>, <sendM, ai_{20}, o>);
addArc({<ao_{21}, recM, i>, <sendM, ai_{15}, o>);
addArc({<ao_{23}, recM, o>, <sendM, ai_{16}, i>);
addArc({<ao_{24}, recM, o>, <sendM, ai_{14}, i>);
newColor(ai_{15}) = freezeInfo;
newColor(ao_{15}) = confirmFreeze;
newColor(ai_{16}) = unfreezeInfo;
newColor(ao_{16}) = confirmUnfreeze;
newColor(E(arc<unfreeorder, ap_{14}, i>)) =
      ((pay is success)and(order is unfrozen);
flush(isolating_pattern);
delNode(isolating_pattern);
```

According to the mismatch pattern, the meta-program provides a suitable adapter realized as a list of actions by using the operators in Table 3, then the meta-program sets the local influence area of the strategy, i.e., a portion of the base-level Petri net that should be changed by the base-level program and need to be frozen during the changes. After applying the evolutionary strategy, every two incompatible services are composed through the adapter as shown

Mismatch Pattern	Operator	Operator Description
Signature mismatch	`Equ`	*The source message is translate into the message requested by the target interface.*
Ordering mismatch	`Flow`	*The source message is reordered to fit the target interface.*
Extra message mismatch	`Hide`	*The extra data in the source message are hidden to match the target interface.*
Missing message mismatch	`Null`	*The source message is padded with null data to match the target interface.*
$1 \rightarrow n$ message mismatch	`Scatter`	*The source message is divided into several to match the target interface.*
$n \rightarrow 1$ message mismatch	`Gather`	*Several source messages are collapsed into a single message matching the target interface.*

Table 3: Transformation Operators used in adapter

in Fig. 5. Adapter is also implemented as a service to compose in the system.

6. RELATED WORKS

Recent development of service-oriented computing and grid computing has led to the rapid adoption of service-oriented architectures in distributed computing systems. Unique characteristics of SOA, such as loosely coupling and late binding, provide the capabilities that enable the rapid composition of the needed services from various service providers for distributed applications and the runtime adaptation of service-oriented architectures [18]. The system needs to be updated or extended with new characteristics during the process of adaption, that is to say the system needs to be self-evolving without stopping and by directly patching the software. A good evolution is carried out through evolution of system design information, and then through propagation of evolution to implementation [6]. However, the evolution is supported by a few of design/specification formalisms at present.

Mazzara *et al.* [11] puts forward a formalism for the modeling and analysis of dynamic reconfiguration of dependable real-time systems. It presents requirements that the formalism must meet, and use these to evaluate well established formalisms and two process algebras that they have been developing, namely, Web$_{\Pi_\infty}$ and CCSdp. It shows a good example to represent a significant step forward in modeling adaptive and dependable real-time systems. However, the above modeling methods only focus on reconfiguration with interference between application activities and reconfiguration activities in dependable real-time systems. When we use this method to model evolving systems, the software designer needs to define both normal and erroneous cases in the reconfiguration manager and cannot express the dynamic behavior for the evolution.

Wörzberger *et al.* [17] propose to simulate on a Static BPMS approach. The approach extends the static BPMS in order to support dynamic changes of processes during execution. It represents the dynamic modifications of workflows as Add, Remove and Iteration, which can be realized by an additional dynamic layer based on existing static BPMS. The approach uses the Web Services Business Process Execution Language (WS-BPEL) constructs types at run-time to modify the static BPMS. The main limitation of this approach is that it uses a small subset of all WS-BPEL construct types and deal with problems arising from the concurrency and distribution of workflows.

Biermann *et al.* [3] puts forward Reconfigurable Object Nets (RONs), which are the integration of transition firing and rule-based net structure transformation of place/transition nets during system simulation. RONs are high-level nets with two types of tokens: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules). Firing of high-level transitions may involve firing of object net transitions, transporting object net tokens through the high-level net, and applying net transformation rules to object nets. Net transformations include net modifications such as merging or splitting of object nets, and net refinement. This approach increases the expressiveness of PNs and is especially suited to model mobile distributed processes.

7. CONCLUSIONS

As service oriented architectures are more and more applied, its evolutionary aspects have been widely recognized as one of crucial challenges. Reflective Petri nets are a formal method that could simulate its evolving process to assure reliable running. But when using reflective PNs to model service composition systems, the dynamic interaction behavior of the system could not be easily modeled, the expressiveness of traditional place/transition PNs is too limited to easily deal with the complexity of the message structure necessary for service interaction. As a result, we propose to use colored petri nets to model service composition system in base-level and adjust the reflective Petri net approach to support it accordingly. Considering the interface mismatch problem in service composition system, we define the meta-program generic schema in Colored RPN. In order to give effective evidence, we show a case study. The proposed approach solves the dynamic composition problem for the evolution of service oriented architectures.

Acknowledgments

This work has been supported by National High-Tech Research and Development Plan of China (No. 2009AA01Z122) and National Natural Science Foundation (No. 61073062, 61100027) and by MIUR DISCO (Distribution, Interaction, Specification, Composition for Object Systems).

8. REFERENCES

[1] Danilo Ardagna and Barbara Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, June 2007.

[2] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing Adapters for Web Services Integration. In João Falcão e Cunha and Oscar Pastor, editors, *Proceedings of the 17th International Conference on Advanced Information Systems Engineerin (CAiSE'05)*, LNCS 3520, pages 415–429, Porto, Portugal, June 2005. Springer.

[3] Enrico Biermann, Claudia Ermel, Frank Hermann, and Tony Modica. A Visual Editor for Reconfigurable Object Nets based on the ECLIPSE Graphical Editor Framework. In *Procedings of 14th Workshop on Algorithms and Tools for Petri Nets (AWPN'07)*, Landau, Germany, October 2007.

[4] Lorenzo Capra and Walter Cazzola. Self-Evolving Petri Nets. *Journal of Universal Computer Science*, 13(13):2002–2034, December 2007.

[5] Lorenzo Capra and Walter Cazzola. An Introduction to Reflective Petri Nets. In Evon M. O. Abu-Taieh and Asim A. El Sheikh, editors, *Handbook of Research on Discrete Event*

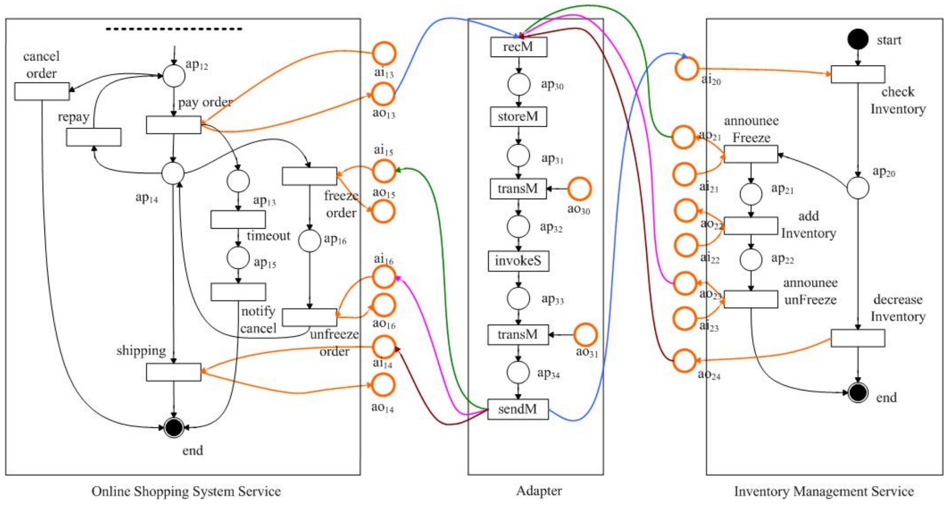

Figure 5: The new on-line shopping system.

Simulation Environments: Technologies and Applications, chapter 9, pages 191–217. IGI Global, November 2009.

[6] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science 2975, pages 69–84. Springer-Verlag, Heidelberg, Germany, July 2004.

[7] Rachid Hamadi and Boualem Benetallah. A Petri Net-Based Model for Web Service Composition. In Klaus-Dieter Schewe and Xiaofang Zhou, editors, *Proceedings of the 14th Australasian Database Conference (ADC'03)*, pages 191–200, Adelaide, Australia, 2003. Australian Computer Society.

[8] Kurt Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *A Decade of Concurrency, Reflections and Perspectives*, LNCS 803, pages 230–272. Springer, June 1993.

[9] Roberto Lucchi and Manuel Mazzara. A Pi-Calculus Based Semantics for WS-PEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[10] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[11] Manuel Mazzara and Anirban Bhattacharyya. On Modelling and Analysis of Dynamic Reconfiguration of Dependable Real-Time Systems. In *Proceedings of the 3rd International Conference on Dependability (DEPEND'10)*, pages 173–181, Venice, Italy, July 2010. IEEE.

[12] Mike P. Papazoglou. The Challenges of Service Evolution. In Zohra Bellahsène and Michel Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, LNCS 5074, pages 1–15, Montpellier, France, June 2008. Springer.

[13] Stephanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness Criteria for Dynamic Changes in Workflow Systems – A Survey. *Data & Knowledge Engineering*, 50(1):9–34, July 2004.

[14] Seung Hwan Ryu, Fabio Casati, Halvard Skogrud, Boualem Benatallah, and Régis Saint-Paul. Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures. *ACM Transaction on the Web*, 2(2), April 2008.

[15] C. Sloan, John and Taghi M. Khoshgoftaar. From Web Service Artifact to a Readable and Verifiable Model. *IEEE Transaction on Services Computing*, 2(4):277–288, October-December 2009.

[16] Wei Tan, Yushun Fan, and Meng-Chu Zhou. A Petri Net-Based Method for Compatibility Analysis and Composition of Web Services in Business Process Execution Language. *IEEE Transactions on Automation Science and Engineering*, 6(1):94–106, January 2009.

[17] René Wörzberger, Nicolas Ehses, and Thomas Heer. Adding Support for Dynamics Patterns to Static Business Process Management Systems. In *Proceedings of the 7th international conference on Software composition (SC'08)*, pages 84–91, Budapest, Hungary, July 2008. ACM, Springer-Verlag.

[18] Stephen S. Yau, Nong Ye, Hessam S. Sarjoughian, Dazhi Huang, Auttawut Roontiva, Mustafa Gökçe Baydogan, and Mohammed A. Muqsith. Toward Development of Adaptive Service-Based Software Systems. *IEEE Transactions on Services Computing*, 2(3):247–260, July-September 2009.

[19] Xiaochuan Yi and Krys J. Kochut. A CP-nets-based Design and Verification Framework for Web Services Composition. In *Proceedings of the 2nd IEEE International Conference on Web Services (ICWS'04)*, pages 756–760, San Diego, CA, USA, July 2004. IEEE.

[20] Jin Zeng, Hai-Long Sun, Xu-Dong Liu, Ting Deng, and Jin-Peng Huai. Dynamic Evolution Mechanism for Trustworthy Software Based on Service Composition. *Journal of Software*, 21(2):261–276, February 2010.

[21] Liangzhao Zeng, Boualem Benatallah, Anne H.H̃. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-aware middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.