IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 8, 2018, accepted January 4, 2019, date of publication January 11, 2019, date of current version February 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2892082

Integrating UML With Service Refinement for Requirements Modeling and Analysis

YILONG YANG[1], WEI KE[2], JING YANG[3], AND XIAOSHAN LI[1]
[1]Faculty of Science and Technology, University of Macau, Macau 999087, China
[2]Macau Polytechnic Institute, Macau 999087, China
[3]College of Computer Science and Technology, Guizhou University, Guiyang 550025, China

Corresponding authors: Jing Yang (yangjing6646@yahoo.com.hk) and Xiaoshan Li (xsl@umac.mo)

This work was supported in part by the projects of the Macau Science and Technology Development Fund under Grant 103/2015/A3, in part by the University of Macau under Grant MYRG-2017-00141-FST, and in part by the National Natural Science Foundation of China under Grant 61562011.

ABSTRACT The Unified Modeling Language (UML) is the *de facto* standard for requirements modeling and analysis in the software industry. However, it lacks the ability of formal analysis and verification. In this paper, we propose a synthetic approach UML-SR that integrates UML with service refinement (SR) to support the formal requirements modeling and analysis as well as formal verification. The UML-SR requirements model contains a use case diagram, the system sequence diagrams of use cases, a conceptual class diagram, and the formal contracts of system interfaces. To make this integration viable, we extend service refinement with the concepts of visibility in UML. With the visibility extension, we are able to formally specify and verify both internal and external interactions of the system. To demonstrate the effectiveness of our proposed approach, we investigate a case study of an *Online Shopping System*. The results show that a consistent requirements model can be eventually derived through formal refinement and verification. The proposed approach is useful and can be further applied for the requirements modeling and formal verification in the software industry.

INDEX TERMS UML, requirements modeling, service refinement, formal verification.

I. INTRODUCTION

One of the major challenges in software development is to conquer the high complexity of current large-scale software system [1], [2]. To tackle this challenge, UML provides various diagrams for developers to model software systems from multiple viewpoints. Under such an approach, the requirements model of a system can be specified by a use case diagram, a set of system sequence diagrams for the interactions between the system interfaces and the environment in each use case, and a conceptual class diagram for modeling the concepts in the target domain. However, the potential issues of the requirements model are hard to find only depending on these diagrams, because UML does not have any technique to verify safety properties, such as deadlock-free and livelock-free in requirements models. Moreover, when the system derived from a requirements model is getting refined, we cannot verify the correctness of the refinement by UML alone. The objective of this paper is to discover the correspondence between UML and service refinement, so as to take the advantages of both UML and formal methods in requirements modeling, with the capability of doing formal safety checking and refinement verification by some popular requirements modeling tool.

Service refinement [3] is a well-designed formal method on the calculus of contract refinement. It has the capability to reason important safety properties of system behaviors and the correctness of refinement. Service refinement lays the foundation of guard design and design refinement on the Unifying Theories of Programming (UTP) [4], and the divergence and failures model of the Communicating Sequential Processes (CSP) [5]. This puts it under our main consideration as the underpinning formal method for UML requirements models. On the other hand, service refinement is also simple enough to let us focus only on requirements modeling and analysis, without going into the design model constructs such as class diagrams, sequence diagrams, collaboration diagrams and state machines. With the highly similar concepts of interfaces in both service refinement and UML, service refinement is much closer to bridge with UML requirements modeling than other potential formal

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

methods based on automata, such as Event-B [6] and UPPAAL [7].

However, service refinement is not natively designed for requirements modeling and analysis, at least not in a fully object-oriented style. In particular, an interface in service refinement consists of only "public" services. To handle those private methods in a UML requirements model, we must extend service requirement with the addition of visibility specifications, i.e., a "public" or "private" prefix can be added to a service. We summarize the adjustments based on this visibility extension to service refinement as follows.

Visibility The visibility prefixes are used to describe whether an operation of an interface is public or private in UML. Correspondingly, we extend service refinement to allow both public and private services in an interface. We further restrict that a use case must go through its private services to invoke services in other use cases, so that all interactions between use cases can be formulated as invocations of the private services of the invoking use cases.

Divergence The divergence state is defined as the unstable state of livelocks, in which the system infinitely invokes internal actions. With the above visibility extension of service refinement, the divergence state is defined as infinite invocations of private services.

Consistency Without the visibility extension, the consistency in service refinement only considers if there are deadlocks. Now, we extend the consistency of contract to take livelocks into account as well. A contract is consistent not only when there exist some services outside the refusal set after a trace, i.e., *deadlock-free*, but also there exists no trace containing infinite invocations of private services, i.e., *livelock-free*.

Refinement A refined contract maintains the same behavior to the environment, but is safer in terms of consistency. For private services which are invisible to the environment, we need to *hide* them when considering the external behavior of a contract so as to make justifications to the refinement relation.

Contribution: The purpose of this work is to provide formal support to UML through service refinement. In order to conduct consistency checking and refinement verification on the contracts derived from a requirements model, we present the bridging from a UML requirements model to the corresponding interfaces and contracts in service refinement by the case study of an Online Shopping System. Our major contribution in this work is summarized as follows.

1) We extend service refinement with visibility prefixes, and formulate the needed adjustments to the theory.
2) We propose a synthetic approach UML-SR by integrating UML with service refinement.
3) We demonstrate the effectiveness of UML-SR for requirements modeling and verification through the case study.

The remainder of this paper is organized as follows. Section II presents the preliminary of service refinement with the extension of visibility prefixes in a formal style. Section III presents UML-SR by integrating UML and service refinement to develop the requirements model, and derives the contracts through the case study of an Online Shopping System. Section IV refines the contracts with the consistency properties and presents the verification of the refinement. Section V reviews the related work. Finally, section VI concludes this paper and discusses the future work.

II. SERVICE REFINEMENT AND VISIBILITY EXTENSION

The theory of service refinement describes how to specify the services of a system and the refinement relations between various concepts in the specifications. The concepts include *interfaces* to describe the signatures of services and the global variables, *specifications* to describe the functionality and behavior, and *contracts* to describe the interaction protocols, in particular, the intended invocation sequences of the services in a system. With the specifications of behaviors and the protocols of interactions, the *consistency* of a system can be defined and reasoned about, where deadlock-free and livelock-free are the properties to deal with. Now, we give the definitions of these concepts in detail with the adjustments introduced by the visibility extension.

A. INTERFACES

Interfaces are the access points of a system. In service refinement, an interface I is a pair consisting of a resource declaration sector and a service declaration sector,

$$I = (RDec, SDec). \tag{1}$$

$RDec$ is a set of variable declarations $\{\overline{x : T}\}$, with x the variable and T the type. $SDec$ is a set of service signatures

$$\{\overline{m(in : U, out : V)}\},$$

with m the name of the service, in of type U the input parameter, and out of type V the output parameter.

1) EXTENSION FOR INTERFACES

A visibility prefix can be added to a signature to indicate whether the service is visible to the environment. We adopt the same visibility notation of UML interfaces, where the "+" prefix indicates a public service, and the "−" prefix indicates a private service. For example, the signature of a private service m is written as

$$-m(in : U, out : V).$$

$Private(SDec)$ is the set of private services, $Public(SDec)$ is the set of public services. By default, a service signature without a prefix is regarded as public. Private services are not visible to the environment, they can only be seen in the contract containing the interface.

Note that service refinement follows the convention of UTP to specify the requirements model:

- The variable x stand for the initial value of a variable updated by the program.
- The decorated version x' represents the final value of that variable on termination.
- Boolean variable ok (and its decorated version ok') takes the value *true* just when the program has reached a stable and observable state.
- Boolean variable $wait$ takes the value *true* just when the program has reached a stable intermediate state, and which is *false* if the program has terminated.

Like in regular expressions, an asterisk (A^*) superscript indicates zero or more repetitions of event A and a question mark ($A^?$) superscript indicates zero or one occurrence of event A. Sequentially invoking the services m_1, m_2, m_3 actually results the trace set

$$\{\langle\rangle, \langle m_1\rangle, \langle m_1, m_2\rangle, \langle m_1, m_2, m_3\rangle\}.$$

B. SPECIFICATIONS

The specification of a service m is a triple $(\alpha, g, P)_m$, where

- m is the name of the service,
- α comprises all the resources managed by the service,
- g is the firing condition of the service, characterizing the circumstance under which the service can be activated,
- P is a reactive design $H(p \vdash R)$, describing the behavior of the service when being executed.

Obviously, a specification does not have any visibility issue.

C. CONTRACTS

A contract is the specification of an interface. A contract of an interface specifies the functionality of each service declared in the interface, and the protocol of the interactions between the services.

A contract Ctr is a quadruple $(I, Ini, Spec, Prot)$, where

- I is an interface,
- Ini specifies the initial design, i.e., for the set of variables V in the resource of I
$$Ini = true \vdash ini(V') \wedge \neg wait',$$
- $Spec$ is the set of specifications $\{\overline{(\alpha, g, P)_m}\}$ for each service $m(in : U, out : V)$ in I, and
- $Prot$ is a set of valid event traces of service requests and responses in the form of
$$\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k), !m_k(y_k)\rangle,$$
where $?m(x)$ represents a request of service m with parameter x and $!m(y)$ the response of service m with result y. The protocol of a contract consists of all the possible request prefixes of a normal response, i.e., if the services are invoked following the traces in $Prot$, we can expect a result, otherwise, the result is undefined.

1) SEMANTICS OF CONTRACTS

The dynamic behavior of a contract Ctr forms the semantics, which is described as a triple

$$(Prot(Ctr), Failures(Ctr), Divergences(Ctr)),$$

where

- $Prot(Ctr)$ retrieves the weakest protocol of the contract,
- $Failures(Ctr)$ is the set of pairs $\{\overline{(s, X)}\}$, where s is a trace of interactions between Ctr and the environment, and X denotes the set of requests and responses which the contract may refuse to engage after the occurrence of trace s, and
- $Divergences(Ctr)$ is the set of traces each of which leads the contract to a divergence.

2) EXTENSION FOR SEMANTICS OF CONTRACTS

In UML, private services of a interface are invisible from the environments, they only can be invoked by the public services co-located in the same interface, and the intermediate variables and boolean variables ok' and $wait'$ cannot be observed during the invoking process. Therefore, when a public service $m \in Public(SDec)$ invokes the private services $\{-m \in Private(SDec)\}$, the traces and observations can be denoted as:

$$\{\langle ?m(x_i), ?{-m}^*\rangle \;|(g_m \;\&\; P_m)[x_i/x][true/ok, true/wait']\}$$

Note that the ok and $wait$ are observed before invoking the private services. At that moment, the system is in a waiting state, i.e, the system waits for private services to be executed. The new values can only be observed when the system finishes invoking the private services. After successfully invoking the private services and returning the results, the traces and observations can be denoted as:

$$\{\langle ?m(x_i), ?{-m}^*, !m(y_i)\rangle \mid (g_m \;\&\; P_m)[x_i/x, y_i/y'][true/ok, true/ok']\}$$

In the relation to UML, $Divergences(Ctr)$ specifies the interaction traces with a use-case that can lead to a divergence of the system, which means the use case are busy in executions without termination. In this case, the divergence of a contract can be defined as the set of traces of infinite sequences of invocations to the private services, denoted by $\{\langle ?{-m}^*\rangle \mid -m \in Private(SDec)\}$. Therefore, the extension is defined as,

$$Divergences(Ctr) =_{df} \left\{ \begin{array}{l} \langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k), ?{-m}^*\rangle \\ \mid Ini; (g_1 \;\&\; P_1)[x_1/x, y_1/y']; \ldots; \\ \quad (g_{k-1} \;\&\; P_{k-1})[x_{k-1}/x, y_{k-1}/y'] \\ \qquad\qquad [true/ok', false/wait']; \\ \quad (g_k \;\&\; P_k)[x_k/x][true/ok, false/ok'] \end{array} \right\}$$

where $(\alpha_i, g_i, P_i)_{m_i} \in Spec(Ctr)$ for $i = 1, 2, \ldots, k$.

As we have mentioned, private services can only be invoked from the public services in the same interface. Therefore, private services are in the refusal set X of $Failures(Ctr)$, except when the public service m_k invokes them, $Failures(Ctr)$ thus have the extended

definition,

$$
\begin{aligned}
&Failures(Ctr) =_{df} \{\ldots\} \\
&\cup \left\{\begin{array}{l}
(\langle ?m_1(x_1), !m_1(y_1), \ldots \\
\quad ?m_{k-1}(x_{k-1}), !m_{k-1}(y_{k-1})\rangle, X) \\
\mid \exists v', y' \cdot (Ini; \ldots; \\
\qquad (g_{k-1} \,\&\, P_{k-1})[x_{k-1}/x, y_{k-1}/y'] \\
\qquad\qquad [true/ok', false/wait'] \\
\qquad \wedge \forall ?m \in X \cdot \neg g[v'/v] \\
\qquad \wedge\ m \in Public(SDec) \\
\qquad \wedge\ ?{-m} \in X \wedge -m \in Private(SDec))
\end{array}\right\} \\
&\cup \left\{\begin{array}{l}
(\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k)\rangle, X) \\
\mid \exists v' \cdot (Ini; \ldots; \\
\qquad (g_{m_k} \,\&\, P_{m_k})[x_k/x] \\
\qquad\qquad [true/ok', true/wait'] \\
\qquad \forall ?m \in X\ \wedge\ m \in Public(SDec) \\
\qquad \wedge\ ?{-m} \notin X \wedge -m \in Private(SDec))
\end{array}\right\}
\end{aligned}
$$

where

$$(\alpha, g, P)_m, \quad (\alpha_i, g_i, P_i)_{m_i} \in Spec(Ctr),$$

for $i = 1, 2, \ldots, k$, and $\{\ldots\}$ denotes the $Failures(Ctr)$ set defined in the original service refinement.

D. CONTRACT CONSISTENCY

A contract Ctr is consistent, if it will never enter deadlock states unless the environment violates the protocol. This can be formally defined as,

$$\forall tr \in Prot \cdot \left(\begin{array}{l} \exists s \in Traces(Ctr) \cdot s_{\downarrow\{?\}} = tr \wedge \\ \forall (s, X) \in Failures(Ctr) \cdot s_{\downarrow\{?\}} \preceq tr \\ \Rightarrow X \neq \{?m, !m \mid m \in SDec\} \end{array}\right) \tag{2}$$

where $Traces(Ctr) =_{df} \{s \mid \exists X \cdot (s, X) \in Failures(Ctr)\}$, is the set of all traces each with a refusal set defined, $s_{\downarrow\{?\}}$ is the filtered subsequence of s keeping only the requests, and the $t \preceq s$ partial order means sequence t is a prefix of s.

EXTENSION OF CONTRACT CONSISTENCY

With the visibility extension and our enforced use of private services in internal interactions between use cases, we are able to well-define the semantics of divergence. Thus, we can also extend the consistency to take livelock-free into account. A contract is consistent when it is both deadlock-free and livelock-free. The former means after any trace there are the services to continue with, i.e., the refusal set X does not contain all the private and public services. Since the extended service declaration sector $SDec$ includes both private and public methods, the deadlock-free condition is covered by (2). The latter means there should not be any infinite requests to private services in any trace:

$$\forall s \in Traces(Ctr) \cdot s_{\downarrow\{?\}} \notin \{t \frown \langle ?{-m}^* \rangle \mid -m \in Private(SDec),\ t \preceq s_{\downarrow\{?\}}\}$$

E. CONTRACT REFINEMENT

A refined contract provides the same set of services but with more safer behaviors.

Let $Ctr_i = (I_i, Ini_i, Spec_i, Prot_i)$, for $i = 1, 2$, be two contracts with the same set of services. We say Ctr_1 is refined by Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if

1) $Divergences(Ctr_1) \supseteq Divergences(Ctr_2)$, and
2) $Failures(Ctr_1) \supseteq Failures(Ctr_2)$.

EXTENSION FOR CONTRACT REFINEMENT

With the visibility extension, the traces of divergences and failures contain invocations of private services. However, from the viewpoint of the environment, these events of private services are invisible. We follow the service hiding to hide all the private services from the traces and refusal sets before we check the above conditions of contract refinement. This works for the traces that lead to divergence and failure, in other words, these are the traces in the blacklist. If we blacklist a trace tr with the private services concealed, we will effectively blacklist all those traces with tr as the result of the concealment.

Let $Ctr_i = (I_i, Ini_i, Spec_i, Prot_i)$ for $i = 1, 2$, be two contracts with the same public services, $-M_1$ and $-M_2$ the respective private services in Ctr_1 and Ctr_2. Ctr_1 is refined by Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if

$$
\begin{aligned}
Divergences(Ctr_1 \setminus -M_1) &\supseteq Divergences(Ctr_2 \setminus -M_2) \\
Failures(Ctr_1 \setminus -M_1) &\supseteq Failures(Ctr_2 \setminus -M_2)
\end{aligned}
$$

where ($\setminus$) is the hiding operator on traces and the set difference on sets of events, also the operation is distributed into sets and pairs when needed without ambiguity. Formally, Let Ctr be a contract, and $-M$ is the private services of $SDec$, $Ctr \setminus -M$ removes the private services of $-M$ from the contract Ctr:

$$
\begin{aligned}
&Divergences(Ctr \setminus -M) =_{df} \{s \mid \\
&\qquad s \in Divergences(Ctr) \\
&\qquad \wedge s \in \{?m(x), !m(y) \mid m \in SDec \setminus -M\}^* \\
&Failures(Ctr \setminus -M) =_{df} \{(s, X) \mid \\
&\qquad (s, X) \in Failures(Ctr) \\
&\qquad \wedge\ s \in \{?m(x), !m(y) \mid m \in SDec \setminus -M\}^* \\
&\qquad \wedge X \subseteq \{?m \mid m \in SDec \setminus -M\}
\end{aligned}
$$

III. REQUIREMENTS MODELING IN UML-SR

In this section, we present how to integrate service refinement with UML as UML-SR for requirements modeling and analysis shipped with a case study of an Online Shopping System.

A. OVERVIEW

The elements of UML and service refinement used in requirements modeling, together with their correlation, are shown in Fig. 1. A requirements model of UML contains a use case diagram, system interfaces, system sequence diagrams and a conceptual class diagram. The corresponding constructs

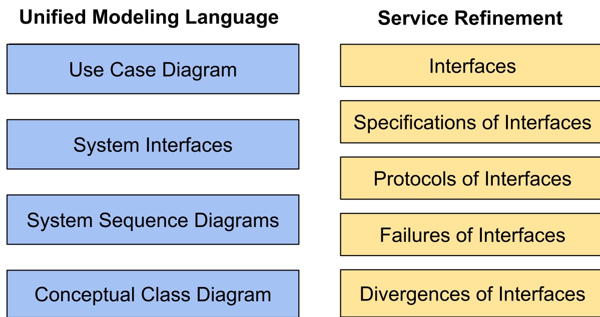

FIGURE 1. Comparison UML with service refinement for requirements modeling.

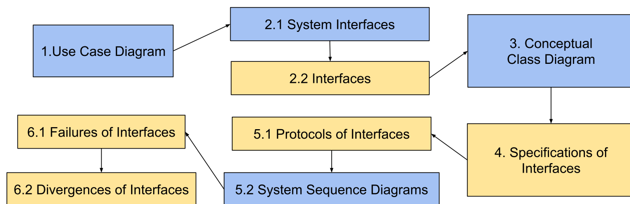

FIGURE 2. The synthetic approach UML-SR for requirements modeling.

in service refinement include interfaces, specifications of interfaces, protocols of interfaces, failures of interfaces and divergences of interfaces.

The overall of UML-SR is shown in Fig. 2. In UML-SR, 1) we adopt a use case diagram to specify the actors, the use cases, and the relations between them in the target system. 2) We use the system interfaces to specify the system operations in each use case and convert the UML interfaces to the formal interfaces in service refinement. 3) We use a conceptual class diagram to specify the entities and their relations of target system based on the specification of resources and services in service refinement. 4) We specify the specifications of interfaces based on step 1, 2 and 3. 5) We use protocols to specify the interactions between the interfaces and the actors for each use case, then convert the result to system sequence diagrams. Finally, 6) we specify the failures and divergences of the interfaces.

The Online Shopping System [8], [9] is an electronic commerce system that allows consumers to directly buy products from sellers over the Internet. The Alibaba, Amazon, and eBay are the largest companies providing online shopping services for billions of people all over the world. Consumers can search interested products through the website, which displays all the related products with their prices from different sellers. Moreover, customers can shop online at any time and anywhere by using a wide range of devices, including desktop computers, laptops, tablets and smart phones.

B. USE CASE DIAGRAM

UML provides use case diagrams to help customers specify the requirements of a target system. The basic use cases of the Online Shopping System are specified in Fig. 3. We list the description of the use cases below.

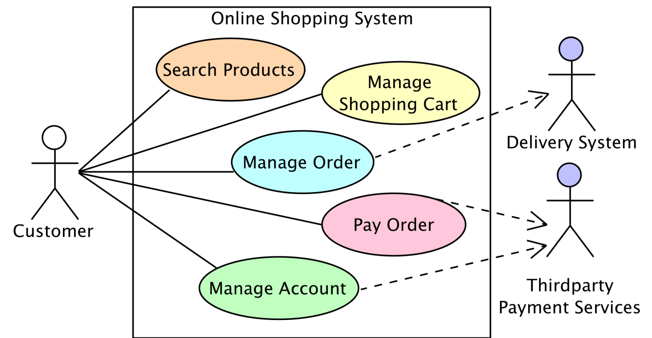

FIGURE 3. Use case diagram of online shopping system.

Search Products The customer can search desired products by keywords, and the system displays the candidate products to the customer for further examinations.

Manage Shopping Cart When customers found the desired products, they can add the products to the shopping cart if the products are available, i.e., in the stock. The customer can also check the state of the shopping cart at any time. While checking the shopping cart, the customer can quickly remove the disliked products from the cart, and the customer can also change the number of items of a product. For example, there is one pair of Nike shoes in Jack's shopping cart, and Jack likes to buy one more pair for his daddy, he could quickly change the number of pairs of the Nike shoes to two in the shopping cart.

Manage Order When the shopping cart is ready, the customer can place the order under his account. This implies the customer must have an account and log into the system. Moreover, at least one address must be added to each account for receiving products. If the customer has more than one addresses, the system will ask to choose one for the shipment. Once a customer has logged into the system, the customer can check the state of the orders at any time. In particular, the system shows whether an order is paid or not. The customer can even track the orders through the delivery system once the products have been sent out.

Pay Order While placing the order, the customer must choose the payment method. If the customer chooses Cash on Delivery, the cash must be given to the delivery man when receiving the products. If the customer chooses the online payment method, the system will check the balance under the account, if the account has not enough balance for the payment, the system must ask the customer to pay with one of the credit cards through a third party payment service.

Manage Account If a customer uses for the first time this Online Shopping System, the customer must register an account before checking orders out. A registered customer can modify the information of the account at any time. In particular, this includes delivery addresses and credit card records. Moreover, a customer can make deposits to the account at any time through third-party payment services.

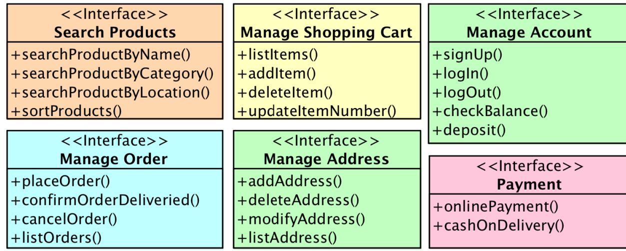

FIGURE 4. Interfaces of online shopping system.

C. SYSTEM INTERFACES AND DOMAIN CONCEPTS MODELING

The primary use cases are presented in the previous section. We now use the interface of the class diagram in UML to specify the services of the use cases in Fig. 4. For example, the *ManageAccountI* interface contains the service *signUp* for a new customer to register the membership, the services *logIn* and *logOut* for a customer to log into and out from the system, the service *checkBalance* for checking balance in the customer's account, and the service *deposit* for increasing the account's balance.

By the interface, we can identify the services of the target system, and even with the input/output parameters and the required resources. However, UML lacks the ability to define the contract for each service to specify precisely how the service should behave. The service refinement formal method not only facilitates precise definitions of service specifications, interfaces and contracts, but also provides the theories to check the consistency properties and verify the refinement of contracts. For example, the *ManageAccountI* interface can be described by the interface of service refinement as follows,

$$
\begin{aligned}
&ManageAccountI = (RDec, SDec)\\
&SDec = \{\\
&\quad signUp(inPersonalInfo : Customer, out : Boolean),\\
&\quad logIn(inUserName : String, inPasswd : String,\\
&\qquad out : Boolean),\\
&\quad logOut(out : Boolean),\\
&\quad checkBalance(outBalance : Double),\\
&\quad deposit(inNewBalance : Double, out : Boolean)\}\\
&RDec = \{\\
&\quad UserDB : Customer^{*},\\
&\quad CurrentCustomer : Customer,\\
&\quad TempCustomer : Customer,\\
&\quad LoginState : Boolean\}
\end{aligned}
$$

The signatures of the services are defined in *SDec*. For example, service *logIn* requires *inUserName* and *inPasswd* of type *String* as input parameters, returns a result *out* of type *Boolean* to indicate whether the customer has successfully logged in. The required resources are defined in *RDec*. For example, the login state of some customer is represented in global variable *LoginState* of type *Boolean*. In particular, the current customer is represented in *CurrentCustomer* of type *Customer*, which is not a basic type such as *Double*, *String*, *Boolean* or *Date*, but a domain concept. This complex type should contain at least the attributes of

$$
\begin{aligned}
(&Name : String, Passwd : String,\\
&\qquad Balance : Double, Address : String)
\end{aligned}
$$

and can be represented as a domain conceptual class in UML. Once we have defined all the interfaces of the system, just like the above interface *ManageAccountI*, we can forge the domain model in a UML conceptual class diagram, as shown in Fig. 5.

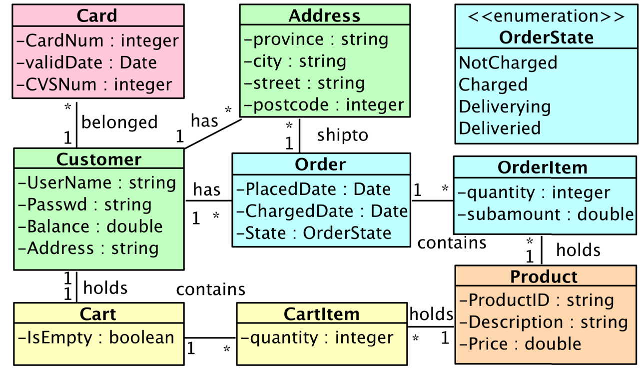

FIGURE 5. Conceptual class diagram of online shopping system.

The conceptual class diagram describes abstract and meaningful concepts in the problem domain, and it decomposes the problem regarding the individual concepts. This is an important trophy in requirement analysis. Therefore, we can tell that service refinement can help UML to get the more precise model about the target system in at least requirement election stage.

D. SERVICE SPECIFICATION BY SERVICE REFINEMENT

We have specified the use cases, the service interfaces, and the domain model of the target system up to this moment. For each service, besides the signature, we also need to precisely define its semantics, such as when the service can be activated, and what the service does (but not how to do) by describing the system state changes carried out by the service. UML diagrams of requirements model cannot describe such semantics of a service, however, the specification triple (α, g, P) in service refinement can be used to accomplish this task. For example, the semantic of service *deposit* can be defined as follows.

$$
\begin{aligned}
&\mathbf{sig} : deposit(inNewBalance : Double, out : Boolean)\\
&\mathbf{spec} : Spec(deposit) = (\alpha, g, p \vdash R)_{deposit} \ \mathbf{where}\\
&\quad \alpha : LoginState, inNewBalance,\\
&\qquad CurrentCustomer, CurrentBalance, out\\
&\qquad \mathbf{let}\ c = CurrentCustomer, \Delta = inNewBalance,\\
&\qquad cb = CurrentBalance\ \mathbf{in}
\end{aligned}
$$

$$g : LoginState = true \land cb \geq 0 \land cb \leq 100$$
$$p : inNewBalance \geq 10$$
$$R : c.Balance' = c.Balance + \Delta \land out' = true \land cb' = c.Balance' \land wait' = false$$

The customer can make a deposit into his account through service *deposit*, which has one input parameter *inNewBalance*, and one output parameter *out*. In the specification, g is a guard condition stating that only when resource *LoginState* is *true* and *CurrentBalance* is less than 100 dollars, the service can be invoked. The design $p \vdash R$ is a pair of precondition p and postcondition R, where p describes that the state of the system before the execution of the service, and R describes the state of the system when the execution of the service completes. For service *deposit* particularly, p specifies the minimum deposit amount which is at least 10 dollars, R specifies after the execution of the service, the balance of current customer is equal to the original balance plus the new deposit amount, and the value of parameter *out* is set to *true*. All the required resources are listed in α including global variables and parameters *LoginState*, *inNewBalance*, *out* and *CurrentCustomer*. Once we have defined the semantics of all the services in the *ManageAccountI* interface, the specification of the interface can be obtained as,

$$Spec(ManageAccountI) = \{Spec(signUp), Spec(logIn), Spec(logOut), Spec(checkBalance), Spec(deposit)\}$$

At this moment, the semantics of all services have been specified. However, the interactions between the interfaces of the system and the environment (actors) are still waiting to be described. These interactions are formulated as the protocol.

E. THE PROTOCOL OF INTERFACE

Service refinement specifies the interactions among actors and use cases in the protocol *Prot* of an interface. The protocol is a set of valid event traces of service requests and responses in the form of

$$\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k), !m_k(y_k)\rangle,$$

where $?m(x)$ represents a request of service m with parameter x and $!m(y)$ the response of service m with result y. For example, the protocol of the *ManageAccountI* interface is defined as follow.

$$\begin{aligned} Prot(ManageAccountI) = \{ s^* \mid\ & s = \langle\rangle \\ \lor\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^?\rangle \\ \lor\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^? \\ & , ?logIn(inUserName, inPasswd), !logIn(out)\rangle \\ \lor\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^? \\ & , ?logIn(inUserName, inPasswd), !logIn(true) \\ & , ?checkBalance(), !checkBalance(outBalance) \\ & , (?deposit(inNewBalance), !deposit(out))^? \\ & , ?logOut(), !logOut(out)\rangle\} \end{aligned}$$

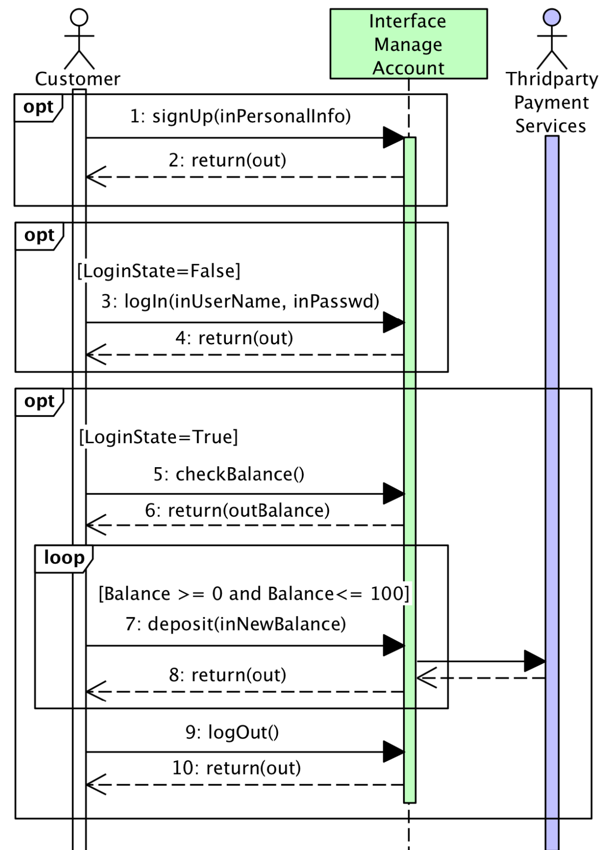

FIGURE 6. The basic event flow of ***ManageAccountI***.

The above protocol of *ManageAccountI* describes four main stories of the interface: 1) In the initial state, no service has been invoked yet, therefore, the trace of the interface is empty. 2) A new customer must request for service *signUp* to open an account in the system. 3) If a customer has successfully opened the account or already has an account in the system, the customer can call service *logIn* to log into the system, then 4) check the balance of the account, deposit into the account if the balance less than \$100, and finally log out of the system.

F. SYSTEM SEQUENCE DIAGRAMS FOR PROTOCOLS

The protocol of an interface can formally define the interactions of use cases as above. However, this notation is not easy to understand by nor intuitive to end-users and developers who are not familiar with formal methods. UML provides activity diagrams and system sequence diagrams that can describe event flows more clearly. For example, the protocol of *ManageAccountI* can be represented by the system sequence diagram show in Fig. 6.

The system sequence diagram illustrates the interface and the actors, and the requests and responses between the interface and the actors. The "**loop**" and "**opt**" operators act exact like (A^*) and $(A^?)$ in the protocol to respectively describe the optional and repeated events. With the

UML system sequence diagram, it is straightforward to formulate the protocol of the interface in the service refinement notation. We also notice that there is a third-part service shown as another actor in Fig. 6. Although we focus on the interactions between a customer and the interface, an actor can also represent other external services in the environment.

G. THE CONTRACT OF INTERFACE

We have just specified the functional requirements in the interface I of the system, the specification *Spec* of the services including the guard condition for activating each service, the state of the system before and after the execution of each service, and the protocol *Prot* of the interactions between the system and the environment. We still lack the information about the initial state of the system. In the theory of service refinement, the contract of interface also specifies the initial state of the resources. For example, for the resources of the $RDec(ManageAccountI)$ below,

$$\begin{aligned}&RDec(ManageAccountI) = \{\\&\quad UserDB : Customer^*,\ CurrentCustomer : Customer,\\&\quad CurrentBalance : Double,\ LoginState : Boolean\}\end{aligned}$$

The initial settings of these resources are described as,

$$\begin{aligned}Ini(ManageAccountI) = true &\vdash \neg wait'\\ \wedge UserDB' &= CustomerDatabase\\ \wedge CurrentCustomer' &= null\\ \wedge CurrentBalance' &= -1\\ \wedge LoginState' &= false.\end{aligned}$$

After system initialization, the resource *UserDB* contains all the customers in the database, *CurrentCustomer* is a reference initialized to *null* meaning this resource does not refer to any customer object, similarly, *CurrentBalance* is -1, and the boolean variable *LoginState* is set to *false*. The *wait* signal is *false* indicating that the system is in a stable state ready to receive service requests from the environment. Adding the initial state of the system finalizes the contract of the interface. As a result, the contract of interface *ManageAccountI* is specified as,

$$\begin{aligned}&Ctr(ManageAccountI)\\&\quad = (ManageAccountI, Ini(ManageAccountI),\\&\quad Spec(ManageAccountI), Prot(ManageAccountI))\end{aligned}$$

1) THE FAILURES OF CONTRACT

The contract of the interface contains the specifications of resources, services, the initial state, and the protocol of the interface. However, the protocol only provides the valid interactions between the system and the environment. For the analysis of safety properties, we not only need to know what interactions the system accepts but also what the system refuses. In service refinement, we can specify the refusal scenarios by modeling the *Failures* of a contract. Recall that a failure is a pair (s, X) where s is a trace of interactions between the system and the environment, and X denotes the refusal set of the services of the contract after the happening of s. The $Failures(Ctr)$ returns the set of all failures of contract *Ctr*, defined as the union of several fragments,

$$Failures(Ctr) =_{df} \left\{\begin{array}{l}(\langle\rangle, X)\mid\\ \exists v'\cdot Ini[true, false/ok, wait]\\ \qquad [true, false/ok', wait']\\ \wedge\forall ?m\in X\cdot\neg g_m[v'/v]\end{array}\right\}\cup\cdots$$

The first fragment of failures describes the refusal of those requests whose guards do not hold in the initial state. For the contract of *ManageAccountI*, the variable *LoginState* is *false* in the initial state. In Fig. 6 showing the workflow of use case *ManageAccountI*, the services *checkBalance*, *deposit* and *logOut* are activated only if $LoginState = true$. This means the mentioned services refuse to response to the environment in the initial state of the system. Formally, after initialization of *ManageAccountI* and the empty trace $\langle\rangle$, the refusal set X is

$$\{?checkBalance, ?deposit, ?logOut\}.$$

The next fragment of the failures is about after the execution of a sequence of requests, including the responses, the guards of some services become *false*, i.e.,

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k), !m_k(y_k)\rangle, X)\mid\\ \exists v'\cdot\ \ (Ini; \ldots; (g_{m_k}\ \&\ P_{m_k})[x_k, y_k/x, y])\\ \qquad [true, false/ok, wait]\\ \qquad [true, false/ok', wait']\\ \qquad \wedge\forall ?m\in X\cdot\neg g_m[v'/v]\end{array}\right\}\cup\cdots$$

For example, the postcondition of the *logIn* service implies $LoginState' = true$ after a successful response. With the guard condition $LoginState' = false$, it means once the customer has logged into the system, the *logIn* service cannot be requested again. Moreover, with the guard condition $CurrentBalance \geq 0 \wedge CurrentBalance \leq 100$, the *deposit* service cannot be requested before getting the current balance because *CurrentBalance* is -1. Formally, after the traces of the *logIn* service,

$$\begin{aligned}&\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\&\qquad ?logIn(inUserName, inPasswd),\ !logIn(out)\,\rangle,\end{aligned}$$

the refusal set X becomes $\{?signUp, ?logIn, ?deposit\}$. Similarly, after the traces of the *logOut* service,

$$\begin{aligned}&\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\&\ ?logIn(\ inUserName, inPasswd), !logIn(out),\\&\ ?checkBalance(), !checkBalance(outBalance),\\&\ (?deposit(inNewBalance), !deposit(out))^*,\\&\ ?logOut(), !logOut(out)\,\rangle,\end{aligned}$$

the variable *LoginState* becomes *false* again. Thus, the refusal set $X = \{?checkBalance, ?deposit, ?logOut\}$.

The next fragment of failures specifies that for every request of a service, the system must have a response. Those traces that refuse to respond, i.e., the response of the last

request is in the refusal set of the trace, must not be included in the failures of the contract, thus, will not be considered as a valid trace of the contract. Formally, the fragment is defined as,

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k)\rangle, X) \mid \\ (Ini; \ldots; (g_{m_k} \ \& \ P_{m_k})[x_k/x]) \\ \quad [true, false, true, false/ok, wait, ok', wait] \\ \wedge \ !m_k \notin X\end{array}\right\} \cup \cdots$$

In Fig. 6 of the event flow of contract *ManageAccountI*, every request event of a service has a corresponding response event. Therefore, the system does not refuse to respond for any request.

2) DEADLOCKS

The next fragment of failures includes the traces that lead the system into waiting states. Formally, the fragment is defined as,

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k)\rangle, X) \mid \\ (Ini; \ldots; (g_{m_{k-1}} \ \& \ P_{m_{k-1}}) \\ \quad [x_{k-1}, y_{k-1}/x, y][true, false/ok', wait']; \\ (g_{m_k} \ \& \ P_{m_k})[x_k/x] \\ \quad [true, false, true, true/ok, wait, ok', wait']\end{array}\right\} \cup \cdots,$$

However, if system is stuck in a permanent waiting state ($wait'$ variable cannot became *false*) and refusal set contains all services of the interface $X = \{?m \mid m \in Public(SDec) \vee Private(SDec)\}$. The system is in the state of deadlock.

For example, the *deposit* service can possibly lead the system into a waiting state, when a customer uses the credit card to add a balance to the account. In this case, the system will wait for the response from the third-party payment service. Usually, the third party service returns instantly and the system sets the *wait* variable to *false* when the *deposit* service completes. However, it is also possible for the third-party service not to return due to communication problems or other external failures. If the requirements are not taking the exceptional cases into account, this will lead to a permanent waiting state, hence a deadlock. The failure of this case can specified as,

$$\begin{array}{l}(\langle(?signUp(inPersonalInfo), !signUp(out))^?, \\ \quad ?logIn(inUserName, inPasswd), !logIn(out), \\ \quad ?checkBalance(), !checkBalance(outBalance), \\ \quad ?deposit(inNewBalance)\rangle, \\ \quad X = \{?m \mid m \in SDec(ManageAccountI)\}).\end{array}$$

This is a terrible property of the system, and we must refine the contract to get rid of the deadlocks before making any implementation. A simple idea to handle this situation is to design an private service *repeatInvokingPayment* to periodically, such as every 30 seconds, send the request to the third-party payment service until there is a response. Note that the *repeatInvokingPayment* must be a deadlock-free service. That means *repeatInvokingPayment* contains a timeout timer, when the timeout period has elapsed without any response, *repeatInvokingPayment* should abandon the future requests and response to the outer caller an error code. This is a common strategy to resolve the indeterministic response time for third party services.

3) LIVELOCKS

The interface of a class diagram can also describe internal services. As shown in Fig. 4, the public services in the UML interface diagram each have a prefix (+), the private services for internal use are prefixed with (−). An internal service does not interact with the environment directly, it is completely hidden from the outside. This complicates the failures of the system in such a way that even if there are invocations to private services within the system, the system may still not be responding. In our *ManageAccountI* example, when third-party payment service is dead, the *repeatInvokingPayment* service will be invoked in infinite times. We say the system is in a livelock, or diverging, if it is busy doing internal services without responding to the environment. When a trace makes the system to fall into a divergence, the system will also refuse to respond to the environment. The final fragment of failures describes this situation. Formally, this fragment is the set of divergent traces defined as,

$$\{(s, X) \mid s \in Divergences(Ctr)\},$$

where $X = \{?{+}m \mid {+}m \in SDec(I(Ctr))\}$. The last event of a divergent trace is a request that leads the system into an unstable state, with the ok' variable is *false*. In a divergence, the internal services are infinitely invoked, that cannot bring the system back to a stable state. In our case, the internal service *repeatInvokingPayment* make system divergent when the third-party service is forever unavailable. Formally, the divergence of interface *ManageAccountI* is,

$$\begin{array}{l}Divergences(Ctr(ManageAccountI)) \\ =_{df} \left\{\begin{array}{l}\langle (?signUp(inPersonalInfo), !signUp(out))^? \\ , ?logIn(inUserName, inPasswd), !logIn(out) \\ , ?checkBalance(), !checkBalance(outBalance) \\ , ?deposit(inNewBalance) \\ , ?repeatInvokingPayment()^*\rangle \\ \mid \ (Ini; \ldots; (g_1 \ \& \ P_1)[outBalance/y]) \\ \quad [true, false/ok', wait']; \\ \ (g_2 \,\&\, P_2)[inNewBalance/x][true, false/ok, ok']\end{array}\right\},\end{array}$$

where

$$\left.\begin{array}{l}(\alpha_1, g_1, P_1)_{checkBalance} \\ (\alpha_2, g_2, P_2)_{deposit}\end{array}\right\} \in Spec(ManageAccountI).$$

Once we specified the divergences of *ManageAccountI*, the final fragment of failures is

$$\{(s, X) \mid s \in Divergences(Ctr(ManageAccountI))\},$$

where $X = \{?{+}m \mid {+}m \in SDec(ManageAccountI)\}$. That is, when a trace s leads to a divergence, all the public services are in the refusal set of s, meaning that the system stops responding to the environment.

In this section, we show how to use service refinement to specify all the possible valid traces from the initial state of an interface, and determine the refusal set of services for each trace. This precisely defines the semantics of the interface. We also see that UML diagrams can help give a clue for modeling each of the use cases. UML diagrams and service refinement are complementary to each other. Although we get all the possible traces and refusal sets, there may still have deadlocks and livelocks. The requirements model must be health before the implementation. That means the requirements are functional correct without the deadlock and livelock.

IV. REFINEMENT AND VERIFICATION IN UML-SR

In this section, we show how to refine the contract of an interface to obtain the deadlock-free and livelock-free properties through service refinement.

A. Ctr_1 IS NOT REFINED BY Ctr_2

The main refinement strategy is to add more control logics and resources to make the refusal set to be a *proper* subset of all the service requests, i.e., $X \subset \{?m \mid m \in SDec\}$ rather than $X = \{?m \mid m \in SDec\}$. As shown in the previous section, the failures of contract *ManageAccountI* contain two traces about the *deposit* service that refuse all the public services afterwards. The first trace is in the deadlock state because of the fault in the request to the third-party payment service. The second one is in the livelock (divergence) state the system infinitely repeats the request to the third-party payment service through the internal $-repeatInvokingPayment$ service. We refer to the original contract of *ManageAccountI* as Ctr_1, and the one with the $-repeatInvokingPayment$ internal service added as Ctr_2. Although Ctr_2 is a deadlock-free contract, it does not really refine Ctr_1, because contract Ctr_2 is more divergent than Ctr_1, i.e., $Divergences(Ctr_2) \supset Divergences(Ctr_1)$ for the additional tail of $?-repeatInvokingPayment^*$ infinite requests in the trace. This violates the refinement condition.

B. Ctr_1 IS REFINED BY Ctr_3

We continue to improve Ctr_2 to be a livelock-free contract Ctr_3 by adding a variable *MaxRepeats* of type *Integer* to $RDec(ManageAccountI)$, with a control logic that once the number of invocations to *repeatInvokingPayment* reaches *MaxRepeats*, e.g., 3 times, it will abandon the request and return a default result $!deposit(false)$. The $wait'$ variable is set to *false* in this case, therefore, the corresponding refusal set will not contain all the public services. After adding the variable *MaxRepeats* and limiting the contract Ctr_3 of interface *ManageAccountI*, when requesting the $?deposit(inNewBalance)$ service, the environment eventually receives the response $!deposit(out)$ after no more than *MaxRepeats* times of invocations to the $-repeatInvokingPayment$ internal service.

$$\begin{aligned}
&\langle(?signUp(inPersonalInfo), !signUp(out))^?\\
&, ?logIn(inUserName, inPasswd), !logIn(out),\\
&, ?checkBalance(), !checkBalance(outBalance)\\
&, ?deposit(inNewBalance)\\
&, ?-repeatInvokingPayment()^{MaxRepeats}\\
&, !deposit(false)\,\rangle
\end{aligned}$$

Ctr_3 will not make system into a divergence state. Therefore, Ctr_3 is a deadlock-free and livelock-free contract, and it refines Ctr_1 because after hiding the events

$$\begin{aligned}-M_3 = \{&?-repeatInvokingPayment,\\ &!-repeatInvokingPayment\}\end{aligned}$$

of the private service, the failures and divergences of Ctr_3 and Ctr_1 satisfy,

1) $Divergences(Ctr_1) \supseteq Divergences(Ctr_3) \setminus -M_3$, and
2) $Failures(Ctr_1) \supseteq Failures(Ctr_3) \setminus -M_3$.

The following is the formal proof of the refinements from contract Ctr_1 to Ctr_3: *Proof:* Ctr_1 is no private service. The failures of contract are:

$$\begin{aligned}
&Failures(Ctr_1) = \{\\
&\quad(\langle\rangle,\\
&\qquad X = \{?checkBalance, ?deposit, ?logOut\}),\\
&\quad(\langle ?signUp(inPersonalInfo), !signUp(out)\rangle,\\
&\qquad X = \{?checkBalance, ?deposit, ?logOut, ?signUp\}),\\
&\quad(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\qquad ?logIn(inUserName, inPasswd), !logIn(out)\rangle,\\
&\qquad X = \{?signUp, ?logIn, ?deposit\}),\\
&\quad(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\qquad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\qquad ?checkBalance(), !checkBalance(outBalance),\\
&\qquad ?deposit(inNewBalance)\rangle\\
&\qquad X = \{?signUp, ?logIn, ?checkBalance,\\
&\qquad\quad ?deposit, ?logOut\}),\\
&\quad(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\qquad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\qquad ?checkBalance(), !checkBalance(outBalance),\\
&\qquad ?deposit(inNewBalance), !deposit(false)\rangle,\\
&\qquad X = \{?signUp, ?logIn, ?checkBalance\}),\\
&\quad(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\qquad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\qquad ?checkBalance(), !checkBalance(outBalance),\\
&\qquad ?deposit(inNewBalance), !deposit(out)\rangle,\\
&\qquad X = \{?signUp, ?logIn, ?checkBalance\}),\\
&\quad(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\qquad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\qquad ?checkBalance(), !checkBalance(outBalance),\\
&\qquad (?deposit(inNewBalance), !deposit(out))^*,\\
&\qquad ?logOut(inNewBalance), !logOut(out)\rangle,\\
&\qquad X = \{?checkBalance, ?deposit, ?logOut\})\\
&\}
\end{aligned}$$

Ctr_3 contains private service $repeatInvokingPayment()$, which only can be invoked by public service $deposit()$. Therefore, the refusal set of $Failures(Ctr_3)$ contains the requests of $?repeatInvokingPayment()$, when the public service $deposit()$ is not executed. The traces of $Failures(Ctr_3)$ contain the requests of $?repeatInvokingPayment()$ representing the repetition of the private requests. The corresponding failures of contract The failures of contract Ctr_3 are,

$$
\begin{aligned}
&Failures(Ctr_3) = \{\\
&(\langle\rangle,\\
&\quad X = \{?checkBalance, ?deposit,\\
&\quad ?logOut, ?repeatInvokingPayment\}),\\
&(\langle ?signUp(inPersonalInfo), !signUp(out)\rangle,\\
&\quad X = \{?checkBalance, ?deposit, ?logOut,\\
&\quad ?signUp, ?repeatInvokingPayment\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out)\rangle,\\
&\quad X = \{?signUp, ?logIn, ?deposit,\\
&\quad ?repeatInvokingPayment\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&\quad (?deposit(inNewBalance),\\
&\quad ?repeatInvokingPayment()^{MaxRepeats},\\
&\quad !deposit(false))^*\rangle,\\
&\quad X = \{?signUp, ?logIn, ?checkBalance,\\
&\quad ?repeatInvokingPayment\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&\quad (?deposit(inNewBalance),\\
&\quad ?repeatInvokingPayment()^{1..MaxRepeats},\\
&\quad !deposit(out))^*\rangle,\\
&\quad X = \{?signUp, ?logIn, ?checkBalance,\\
&\quad ?repeatInvokingPayment\})\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&\quad (?deposit(inNewBalance),\\
&\quad ?repeatInvokingPayment()^{1..MaxRepeats},\\
&\quad !deposit(out))^*,\\
&\quad ?logOut(inNewBalance), !logOut(out)\rangle,\\
&\quad X = \{?checkBalance, ?deposit, ?logOut,\\
&\quad ?repeatInvokingPayment\})\\
&\quad \}
\end{aligned}
$$

Hiding private services $-M_3$ from $Failures(Ctr_3)$ is:

$$
\begin{aligned}
&Failures(Ctr_3) \setminus -M_3 = \{\\
&(\langle\rangle,\\
&\quad X = \{?checkBalance, ?deposit, ?logOut\}),\\
&(\langle ?signUp(inPersonalInfo), !signUp(out)\rangle,\\
&\quad X = \{?checkBalance, ?deposit, ?logOut, ?signUp\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out)\rangle,\\
&\quad X = \{?signUp, ?logIn, ?deposit\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&(?deposit(inNewBalance), !deposit(false))^*\rangle,\\
&\quad X = \{?signUp, ?logIn, ?checkBalance\}),\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&\quad (?deposit(inNewBalance), !deposit(out))^*\rangle,\\
&\quad X = \{?signUp, ?logIn, ?checkBalance\})\\
&(\langle(?signUp(inPersonalInfo), !signUp(out))^?,\\
&\quad ?logIn(inUserName, inPasswd), !logIn(out),\\
&\quad ?checkBalance(), !checkBalance(outBalance),\\
&\quad (?deposit(inNewBalance), !deposit(out))^*,\\
&\quad ?logOut(inNewBalance), !logOut(out)\rangle,\\
&\quad X = \{?checkBalance, ?deposit, ?logOut\})\\
&\quad \}
\end{aligned}
$$

Note that the guard condition of service $checkBalance()$ is $CurrentBalance = -1$. Therefore, it can be only invoked once when logging in to the system. The postcondition of service $deposit()$ indicates that $CurrentBalance$ will be updated after every deposit processes. The difference between the failures of $Failures(Ctr_3) \setminus -M_3$ and $Failures(Ctr_1)$ is that $Failures(Ctr_3) \setminus -M_3$ does not contain the deadlock pairs (4th pair above) in $Failures(Ctr_1)$. Furthermore, both Ctr_3 and Ctr_1 are livelock-free contracts, i.e, $Divergences(Ctr_1) = \emptyset \wedge Divergences(Ctr_3) = \emptyset$. Therefore, Ctr_1 is refined by Ctr_3:

$$
\begin{aligned}
Failures(Ctr_1) &\supseteq Failures(Ctr_3) \setminus -M_3\\
Divergences(Ctr_1) &\supseteq Divergences(Ctr_3) \setminus -M_3
\end{aligned}
$$

□

C. CONTRACT CONSISTENCY

Contract Ctr_3 is a consistent, it will never enter deadlocks and livelocks if the environment follows the protocol of the contract.

Proof: The protocol of Ctr_3 is:

$$Prot(Ctr_3) = \{s^* \mid$$

$$
\begin{aligned}
& s = \langle\rangle \\
\vee\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^{?}\rangle \\
\vee\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out)\rangle \\
\vee\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out) \\
& \quad , ?checkBalance(), !checkBalance(outBalance) \\
& \quad , (?deposit(inNewBalance), \\
& \quad ?repeatInvokingPayment()^{1..MaxRepeats}, \\
& \quad !deposit(out))^{*}\rangle\} \\
\vee\ & s = \langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out) \\
& \quad , ?checkBalance(), !checkBalance(outBalance) \\
& \quad , (?deposit(inNewBalance), \\
& \quad ?repeatInvokingPayment()^{1..MaxRepeats}, \\
& \quad !deposit(out))^{*}, ?logOut(), !logOut(out)\rangle\}
\end{aligned}
$$

The failures of contract Ctr_3 are,

$$
\begin{aligned}
& Failures(Ctr_3) = \{ \\
& (\langle\rangle, \\
& \quad X = \{?checkBalance, ?deposit, ?logOut, \\
& \quad ?repeatInvokingPayment\}), \\
& (\langle ?signUp(inPersonalInfo), !signUp(out)\rangle, \\
& \quad X = \{?checkBalance, ?deposit, ?logOut, ?signUp, \\
& \quad ?repeatInvokingPayment\}), \\
& (\langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out)\rangle, \\
& \quad X = \{?signUp, ?logIn, ?deposit, \\
& \quad ?repeatInvokingPayment\}), \\
& (\langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out), \\
& \quad ?checkBalance(), !checkBalance(outBalance), \\
& \quad (?deposit(inNewBalance), \\
& \quad ?repeatInvokingPayment()^{MaxRepeats}, \\
& \quad !deposit(false))^{*}\rangle, \\
& \quad X = \{?signUp, ?logIn, ?checkBalance, \\
& \quad ?repeatInvokingPayment\}), \\
& (\langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out), \\
& \quad ?checkBalance(), !checkBalance(outBalance), \\
& \quad (?deposit(inNewBalance), \\
& \quad ?repeatInvokingPayment()^{1..MaxRepeats}, \\
& \quad !deposit(out))^{*}\rangle, \\
& \quad X = \{?signUp, ?logIn, ?checkBalance, \\
& \quad ?repeatInvokingPayment\}) \\
& (\langle(?signUp(inPersonalInfo), !signUp(out))^{?}, \\
& \quad ?logIn(inUserName, inPasswd), !logIn(out), \\
& \quad ?checkBalance(), !checkBalance(outBalance), \\
& \quad (?deposit(inNewBalance), \\
& \quad ?repeatInvokingPayment()^{1..MaxRepeats}, \\
& \quad !deposit(out))^{*}, \\
& \quad ?logOut(inNewBalance), !logOut(out)\rangle, \\
& \quad X = \{?checkBalance, ?deposit, ?logOut, \\
& \quad ?repeatInvokingPayment\}) \\
& \}
\end{aligned}
$$

The difference between the failures of Ctr_3 and Ctr_1 is that $Failures(Ctr_3)$ does not contain the deadlock and livelock pairs in $Failures(Ctr_1)$. Therefore, the contract Ctr_3 is a consistent contract, satisfying,

1) $\forall tr \in Prot(Ctr_3) \cdot (\exists s \in Traces(Ctr_3) \cdot s_{\downarrow\{?\}} = tr)$, and
2) $\forall tr \in Prot, \forall(s, X) \in Failures(Ctr_3) \cdot (s_{\downarrow\{?\}} \preceq tr \Rightarrow X \neq \cup\{\{?m, !m\} \mid m \in SDec_3)\} \wedge s_{\downarrow\{?\}} \neq \{t \,\hat{}\, \langle ?{-m}^{*}\rangle \mid -m \in Private(SDec_3), t \preceq s\})$.

Condition 1 says, for every trace in the protocol, there is a corresponding refusal set defined in the *Failures*, describing what services cannot be invoked afterwards. Condition 2 says, for every the trace of the protocol, there must be at least one service that can be invoked afterwards, and the trace does not include infinite invocations to a private services. In other words, a consistent contract must be both deadlock-free and livelock-free. □

V. RELATED WORK

All the related work about service refinement is summarized. The paper [10] proposes a denotational semantics model to service orchestration languages with service refinement, and it can determine whether service orchestration satisfies its specification. The paper [11] based on service refinement proposes a formal model for web service interfaces and mismatch detection among multiple web services. The paper [12] proposes a formal model to specify and analyze the behavior and robustness of service mashups under an unstable environment. The paper [13] uses service refinement to provide a mathematical model for WSDL 2.0. The paper [14] proposes a concept of promoting models to obtain refinements with support from cooperating models. Two papers [15] and [16] are the case studies of this theory but do not include the contract refinement, refinement verification, and consistency checking. To the best of our knowledge, all these efforts do not touch the integration of service refinement and UML requirements analysis to support contract refinement on use cases and consistency checking. Neither do they demonstrate the power of UML in helping specify the start-up requirements for service refinement.

Other formal methods are also considered to be integrated with UML for system modeling and verification.

The paper [17] motivates an approach to formalizing UML in which formal specification techniques are used to gain the insights into the semantics of UML notations and diagrams. A small example is presented through the Z notation to verify whether one class diagram is a valid deduction of another. Another work [18] integrates UML class diagrams and OhCircus by written UML elements in terms of OhCircus constructs. UML-B [19], [20] provides a UML front-end for the B methods and Event-B, where the latter provides a formally precise variant of UML to support model refinement. The paper [21] provides a tool for automatic verification of UML models after transforming the active behavior from UML activity diagrams and class diagrams into SMV. In short, the related work focuses on verifying the design model (class diagrams) of the system specified by UML. Our approach conducts the verification earlier in the stage of requirements analysis, because 1) the problems in a requirements model can be passed to the design model, and 2) to verify a design model, we not only need to specify a requirements model first but also need the effort to specify a design model. To be safe and take less prework, our approach verifies the system in a stage of the process of software engineering as early as possible. This is the main novel idea of our approach.

VI. CONCLUSIONS AND FUTURE WORK

Formal methods provide us a rigorous way to reason about the critical properties of a system, that are otherwise hard to be ensured only by testing. However, formal methods are not so familiar nor very friendly to everyday software engineers and developers. The integration of formal methods and popular modeling tools is a good start to take the advantages of both sides. With our visibility extension to the service refinement, we are able to integrate UML and service refinement for enhancing requirements modeling and analysis. We show how UML requirements modeling can be mapped to constructs in service refinement through a typical case study, an Online Shopping System. Furthermore, we derive the contract of the interface and reason about the saftety properties, namely deadlocks and livelocks. As a result, we use service refinement to obtain a deadlock-free and livelock-free contract.

The case study in our discussion can be generalized to help developers use UML and service refinement together to present a consistent requirements model in the early stage of software development, so that lots of unnecessary bugs can be avoided. Based on the formal analysis of requirements with our proposed method, the contracts can be further realized by object-oriented, component-based, and service-oriented approaches. In the future, we will integrate this formal approach with our developed automatic prototype generation tool RM2PT[1] for requirement validation and evolution.

[1] http://rm2pt.mydreamy.net

REFERENCES

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. London, U.K.: Pearson, 2012.

[2] I. Sommerville, *Software Engineering* (International Computer Science Series). Reading, MA, USA: Addison-Wesley, 2015.

[3] J. He, "Service refinement," *Sci. China F, Inf. Sci.*, vol. 51, no. 6, pp. 661–682, 2008.

[4] C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, 1998.

[5] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, 1985.

[6] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2010.

[7] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, nos. 1–2, pp. 134–152, 1997.

[8] W. Yu, C. G. Yan, Z. Ding, C. Jiang, and M. Zhou, "Modeling and verification of online shopping business processes by considering malicious behavior patterns," *IEEE Trans. Autom. Sci. Eng.*, vol. 13, no. 2, pp. 647–662, Apr. 2016.

[9] D. B. Hopson and K. S. Keys, "Online shopping system," U.S. Patent 7 590 567 B2, Sep. 15, 2009.

[10] Q. Li, H. Zhu, and J. He, "A denotational semantical model for Orc language," in *Theoretical Aspects of Computing—ICTAC*. Berlin, Germany: Springer, 2010, pp. 106–120.

[11] S. Wang, G. Zhang, X. Zhang, and Y. Yang, "A method for detecting behavioral mismatching Web services," in *Proc. 6th Web Inf. Syst. Appl. Conf. (WISA)*, Sep. 2009, pp. 116–121.

[12] Q. Li, J. Shi, and H. Zhu, "A formal framework for service mashups with dynamic service selection," *Innov. Syst. Softw. Eng.*, vol. 10, no. 3, pp. 219–234, 2014.

[13] A. Zhang and X. Xie, "Web services semantic model system," in *Proc. 3rd Int. Conf. Anti-Counterfeiting, Secur., Identificat. Commun. (ASID)*, 2009, pp. 592–595.

[14] Q. Li, Y. Zhao, X. Wu, and S. Liu, "Promoting models," in *Unifying Theories of Programming*. Berlin, Germany: Springer, 2010, pp. 234–252.

[15] J. Liu and J. He, "Reactive component based service-oriented design—A case study," in *Proc. 11th IEEE Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, Aug. 2006, p. 10.

[16] S. Herold *et al.*, "CoCoME—The common component modeling example," in *The Common Component Modeling Example*. Berlin, Germany: Springer, 2008, pp. 16–53.

[17] A. Evans, R. France, K. Lano, and B. Rumpe, "The UML as a formal modeling notation," in *Proc. Int. Conf. Unified Modeling Lang.* Springer, 1998, pp. 336–348.

[18] R. M. Borges and A. C. Mota, "Integrating UML and formal methods," *Electron. Notes Theor. Comput. Sci.*, vol. 184, pp. 97–112, Jul. 2007.

[19] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 92–122, 2006.

[20] C. Snook and M. Butler, "UML-B and event-B: An integration of languages and tools," in *Proc. IASTED Int. Conf. Softw. Eng. (SE)*, Innsbruck, Austria. Anaheim, CA, USA: ACTA Press, 2008, pp. 336–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=1722603.1722663

[21] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente, "UML automatic verification tool with formal methods," *Electron. Notes Theor. Comput. Sci.*, vol. 127, no. 4, pp. 3–16, 2005.

YILONG YANG received the B.S. degree in computer science from the China University of Mining and Technology, China, in 2010, and the M.S. degree from Guizhou University, China, in 2013. He is currently pursuing the Ph.D. degree in software engineering with the University of Macau. He has been a Fellow with the International Institute for Software Technology, United Nations University, Macau. His research interests include automated software engineering and machine learning.

WEI KE received the Ph.D. degree in computer applied technology from Beihang University, in 2012. He is currently an Associate Professor with the School of Public Administration, Macau Polytechnic Institute. His research interests include programming languages, formal methods, software engineering tool support, and software implementation. He had successfully applied in a couple of research projects funded by the Macau FDCT, including the areas of formal methods and software engineering.

JING YANG received the B.S. degree in math from Southwestern Normal University, China, in 1990, and the M.S. and Ph.D. degrees in math and computer science from Guizhou University, China, in 1993 and 2006, respectively. She is currently a Professor with the College of Computer Science and Technology, Guizhou University. Her research interests include mathematical logic and formal method.

XIAOSHAN LI received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, in 1994. He is currently an Associate Professor with the Department of Computer and Information Science, Faculty of Science and Technology, UMAC. His research interests include formal methods, object-oriented software engineering with the Unified Modeling Language, real-time specification and verification, and the semantics of programming language.