Received 27 November 2023, accepted 7 January 2024, date of publication 10 January 2024,
date of current version 19 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3352115

RESEARCH ARTICLE

Typing Requirement Model as Coroutines

QIQI GU AND WEI KE
Faculty of Applied Sciences, Macao Polytechnic University, Macau, China

Corresponding author: Qiqi Gu (qiqi.gu@mpu.edu.mo)

This work was supported in part by the research project funded by Macao Polytechnic University, Macau, under Grant RP/ESAP-02/2017.

ABSTRACT Model-Driven Engineering (MDE) is a technique that aims to boost productivity in software development and ensure the safety of critical systems. Central to MDE is the refinement of high-level requirement models into executable code. Given that requirement models form the foundation of the entire development process, ensuring their correctness is crucial. RM2PT is a widely used MDE platform that employs the REModel language for requirement modeling. REModel contains contract sections and other sections including a UML sequence diagram. This paper contributes a coroutine-based type system that represents pre- and post-conditions in the contract sections in a requirement model as the receiving and yielding parts of coroutines, respectively. The type system is capable of composing coroutine types, so that users can view functions as a whole system and check their collective behavior. By doing so, our type system ensures that the contracts defined in it are executed as outlined in the accompanied sequence diagram. We assessed our approach using four case studies provided by RM2PT, validating the accuracy of the models.

INDEX TERMS Coroutine, model-driven engineering, static analysis, type system.

I. INTRODUCTION

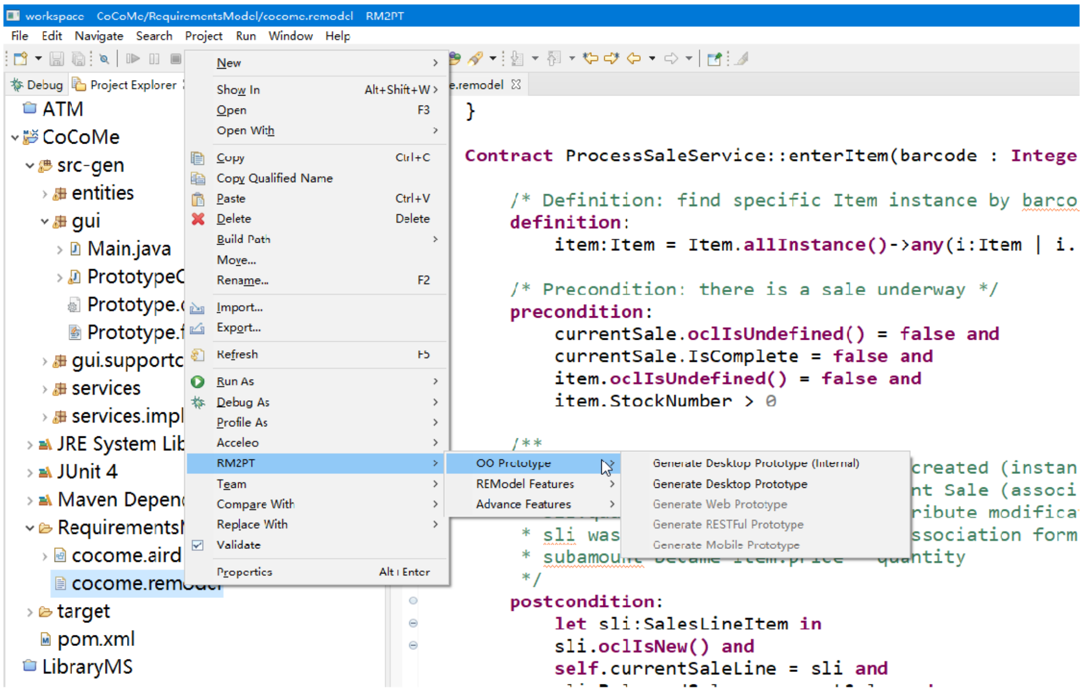
In the realm of software development, Model-Driven Engineering (MDE) has emerged as a popular paradigm for safety critical systems [1], elevating models to the forefront of the process. MDE enables developers to manage the complexity of software by working at a higher level of abstraction and offers the promise of automatic code generation. At the heart of this approach is the refinement of high-level requirement models into design models, and eventually down to executable code—without manual intervention [2]. Such an emphasis on requirement models underscores their paramount importance, for a misstep at this initial model can spell errors for the entire project. Thus, careful requirements modeling, paired with early validation and verification, paves the way for clarity and confidence in the envisioned systems [3], [4]. One form of requirements validation is to ensure the requirements are consistent [5]. We firmly believe that by type checking a requirement model, a high level of consistency can be achieved.

Yang et al. [6] proposed tool RM2PT, which is a powerful and extensible platform that can generate executable prototypes in the Model-View-Controller (MVC) pattern from requirement models automatically. The requirement code RM2PT reads is in REModel format. RM2PT can also visualize requirement code, and the MVC program it generates has GUI for users to view states of the program and check correctness of the model. Fig. 1 exhibits the user interface of the tool. Users can right-click on a.remodel file, and choose to generate a prototype. The editor on the right panel is displaying the file `cocome.remodel`.

RM2PT is a promising project that inspires a number of research projects. For example, Gu et al. [7] transforms the output of RM2PT into smart contracts running on Hyperledger Fabric. Bao et al. [8] generates comments for REModel files, and Yang et al. [9] generates pre- and post-conditions in REModel files from natural languages. Reyal et al. [10] provides guidelines of the UI of the generated prototype.

The REModel language is modified from Object Constraint Language (OCL) [11]. OCL is mainly used for specifying the constraints of UML models including invariants, pre- and post-conditions. REModel extends OCL in that a REModel file can define a collection of contracts with such constraints. Contracts will then be transformed to methods in Java.

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



```
}

Contract ProcessSaleService::enterItem(barcode : Intege

    /* Definition: find specific Item instance by barco
    definition:
        item:Item = Item.allInstance()->any(i:Item | i.

    /* Precondition: there is a sale underway */
    precondition:
        currentSale.oclIsUndefined() = false and
        currentSale.IsComplete = false and
        item.oclIsUndefined() = false and
        item.StockNumber > 0

    /**
                                    created (instan
                                    nt Sale (associ
                                    ribute modifica
     * sli was                      ssociation form
     * subamount                    quantity
     */
    postcondition:
        let sli:SalesLineItem in
        sli.oclIsNew() and
        self.currentSaleLine = sli and
```

FIGURE 1. In RM2PT, Java source code can be generated from a.remodel file through its context menu.

REModel includes other useful functional views for software modeling [12]. For one thing, there is an `Interaction` section specifying the order in which the contracts should be called. This section is rendered as a UML sequence diagram by RM2PT.

Although the RM2PT platform comes with a highlighting service for REModel, the exact syntax of the language is not disclosed, and there is no type checking on requirement code. If the output of one contract does not agree with the input type of another contract, the requirement model is wrong and cannot be used for further development. Hence, to reduce errors in requirement code and make editing REModel files easier, this paper aims to add type checking to REModel files, ensuring pre- and post-conditions match the embedded sequence diagram, and help generate integration test cases. The interplay of pre- and post-conditions in Java methods mirrors the receiving and yielding action of coroutines, and coroutine is a generalization of function, capable of receiving and yielding data more than once.

In this paper, we contribute a type system (the gray box in Fig. 2) for requirement models in MDE. Our type system creatively types contracts in a requirement model as coroutines by using the Typer component in the figure. The second contribution is rules for coroutine type composition, i.e., the Composer. Our rules compose a set of coroutine types into one type to model the collaborative behavior of these coroutines, and permit users to view a set of coroutines as a single coroutine or function. The input to our type system is one REModel file, and the output is a composed type and a yielding order, detailed in Fig. 2.

We firstly formalize the syntax of contracts in REModel with ANTLR 4 [13] lexer and parser rules. Subsequently we put forward typing rules from the REModel language so we can infer coroutine types from the contracts (bags of pre-conditions and post-conditions) defined therein. Lastly, we use the composition rules to compose the inferred coroutine types and check whether the contracts are compatible.

The remainder of this paper is organized as follows. For starters, Section II presents composition rules that are invoked in the compose step in Fig. 2. Then, Section III introduces the syntax of contracts in.remodel files and lists the typing rules, which are the parse step and the type step in the architecture overview. Section IV demonstrates our coroutine type system, and makes use of the composed type and the yielding order. Discussion and limitation is in Section V. Related work is reviewed in Section VI. Finally we conclude.

II. COROUTINE TYPE SYSTEM

This paper contributes a novel type system on the REModel language that types contract blocks in this language as coroutine types, then the coroutine types are composed for verifying the correctness of the requirement model. This section here presents the rules for type composing because it is more interesting. These rules can be hooked or listened, and one application is to formulate a potential execution order. This usage is to be discussed in Section IV-A. Then, Section III formalizes the syntax of REModel and lists typing rules.

The syntax of our type expressions is defined in Fig. 3. A type t can be a concrete type like Int or StringBuilder, a sequence $\langle t_1, t_2 \rangle$, a tuple (t_1, t_2), or a coroutine $[t_1; t_2]$

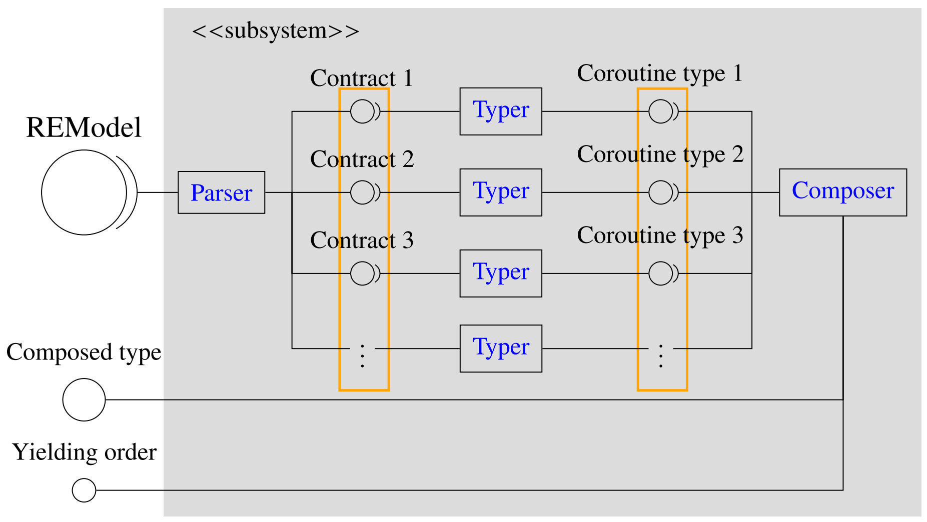


FIGURE 2. The architecture of the requirement model checking framework depicted in the UML component diagram.

$t ::=$		types T
	$\mid t/p$	constrained types
	$\mid \langle t, t \rangle$	sequence types S
	$\mid (t, t)$	product types P
	$\mid [t; t]$	coroutine types R
	$\mid \mathrm{Int} \mid \mathrm{String} \mid \cdots$	concrete types K

FIGURE 3. Abstract syntax for types in our coroutine type system.

where t_1 is named the receiving part and t_2 the yielding part. The two parts are a protocol to control when to execute a coroutine. A coroutine type can take an optional predicate p. For instance,

$$l : \big[\langle x, \mathrm{BookCopy}\rangle ; \langle x, \mathrm{BookCopy}, \mathrm{Reserve}\rangle\big] / x <: \mathrm{User}$$

reads: "l is a coroutine which receives x and BookCopy, and yields x, BookCopy, and Reserve, where x is a subclass of User."

Int, String, and the like are regular types we are familiar with, as seen in expressions like $1 : \mathrm{Int}$ and $\mathrm{foo} : \mathrm{String}$. Then, K along with S, P, R should be perceived as a meta-type akin to the kind $(*)$ in Haskell [14], [15] or Set_1 in Agda [16].

In abstract syntax, p stands for one product type, and its (meta-)type is P, i.e., $p : P$. A coroutine type is presented by θ, and its type is R, i.e., $\theta : R$. A sequence type is presented by Θ, and its type is S.

A sequence is flat, satisfying the associative law, $\langle\langle t_1, t_2\rangle, t_3\rangle = \langle t_1, \langle t_2, t_3\rangle\rangle$, where both sides can be simplified to $\langle t_1, t_2, t_3\rangle$. A sequence of a single type t is called a list of t, written as t^n, where n is the length. A list of indefinite length is denoted as t^*, while a list of zero length is denoted as $\varnothing$. A product functions as a tuple in programming languages.

Many languages have built-in support for product types. Although product types can be seen as a concrete type from the language, we baked in native support of tuples because they can define priorities in composition. In another word, types in a tuple are composed first. Constrained types make the composition less sensitive to the activation order of coroutines. We will discuss the importance of constraints in depth in Section IV-B.

A. COMPOSITION RULES

In this subsection, we introduce the compose function $\circ : S \to R$ that takes a sequence of types and returns a composed coroutine type. The input sequence can contain coroutine types and products of coroutines. Concrete types can only be included in coroutine types R. The return type is R and we do not further reduce it to K, so we know the data are computed rather than from a static field. The compose function $\circ$ consists of rules running in a demand-driven strategy [17].

A coroutine with empty receiving part and empty yielding part is regarded as useless and should be removed from the argument of $\circ$. A sequence cannot contain $\varnothing$ either. Composition Rule CR1 lists the reduction regarding removing these identity elements. $e_1 \Rightarrow e_2$ means an expression e_1 evaluates or reduces to another expression or value e_2.

$$\begin{aligned}\circ(\langle\Theta_1, [\varnothing; \varnothing], \Theta_2\rangle) &\Rightarrow \circ(\langle\Theta_1, \Theta_2\rangle)\\ \circ(\big\langle\Theta_1, \varnothing^*, \Theta_2\big\rangle) &\Rightarrow \circ(\langle\Theta_1, \Theta_2\rangle)\\ \circ(\langle\Theta_1, \varnothing, \Theta_2\rangle) &\Rightarrow \circ(\langle\Theta_1, \Theta_2\rangle)\end{aligned} \tag{CR1}$$

CR2 is particularly useful for establishing priorities for coroutines, ensuring that high-priority coroutines—enclosed in a tuple—are composed first. The reduction applies the function $\mathbf{s}$, which converts a product into a sequence.

$$\circ\langle\Theta_1, p, \Theta_2\rangle \Rightarrow \circ\langle\Theta_1, \circ(\mathbf{s}(p)), \Theta_2\rangle \tag{CR2}$$

In a coroutine, the receiving part must be fully satisfied before the yielding part runs. Interleaving receiving and yielding is impossible. Hence, variables in the receiving part must be bound for later usage in the yielding part. We require to run the receiving part first in order to model the worst

$$\begin{aligned}
\mathbf{h}(k) &= k && (k \in \mathbf{K})\\
\mathbf{h}([s;t]) &= [s;t]\\
\mathbf{h}(\langle s,t\rangle) &= \mathbf{h}(s)\\
\mathbf{t}(k) &= \varnothing && (k \in \mathbf{K})\\
\mathbf{t}([s;t]) &= \varnothing\\
\mathbf{t}(\langle s,t\rangle) &= \langle \mathbf{t}(s), t\rangle
\end{aligned}$$

FIGURE 4. Definition of head h and tail t of a type.

scenario of a function which needs to receive data before starting to yield. As a result, yielding to itself will happen no more.

The rest of composition rules need to access or manipulate auxiliary data. Thus the symbol $\vdash$ is employed. Left to $\vdash$ is the context containing Pending Type t and External Yields E. Right to $\vdash$ is an expression which can be a type or a call to the compose function. We sometimes use a wildcard item $\cdot$ to indicate an unreferenced item in rules. For example $(\cdot, E)$ means at this point the pending type, which may or may not be $\varnothing$, is not used in other parts of the rule.

$$(\varnothing, E) \vdash \circ([s;\, t]) \Rightarrow (\varnothing, \varnothing) \vdash [s;\, \langle E, t\rangle] \qquad \text{(CR3)}$$

In CR3, we start with a context that contains an empty Pending Type. To compose a single coroutine, the yielding part t of the coroutine is prefixed with External Yield E, forming a new sequence $\langle E, t\rangle$. The reason for this ordering is that E represents what other coroutines have already yielded. This rule is a terminal step because on the right-hand side of the $\Rightarrow$ symbol, we don't have another call to $\circ$. The end result is a single coroutine, matching the definition $\circ : S \to R$.

The receiving part and the yielding part can be a sequence. Therefore, we define a head function and a tail function to get the first element and rest elements of a type, shown in Fig. 4. When the yielding part is a sequence, elements are yielded one by one, and the yielded element becomes the pending type. When the receiving part is a sequence, the head of the sequence is the demand, driving the composition. For all other types, the head is themselves and the tail is nothing. To support first-class coroutines, the head of a coroutine type is itself because we are not willing to break the structure. If coroutine a yields coroutine b, we want to yield b as a whole, rather than only yielding part of b.

With the head and tail functions, we are ready to detail the rules with respect to the yield operation and the resume operation.

1) YIELDING

We use a function **first** to find the first coroutine θ in a list Θ of coroutines that matches a condition p, written as $(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.p(\theta))$, where along with the coroutine, **first** also returns Θ_1 all elements before θ and Θ_2 all elements after θ. If **first** cannot find a matching element, $\theta = \Theta_2 = \varnothing$.

CR4, as shown at the bottom of the next page is triggered when there is no Pending Type and the function **first** is able to find the a coroutine whose yielding part is not $\varnothing$. Then, we transfer the head of its yielding part into the context. In case a coroutine has exhausted its action statements, its yielding part $\mathbf{t}(s)$ would be $\varnothing$ and it's subject to deletion by CR1.

Rather than yielding a concrete type, if the yielded type is a coroutine, CR5, as shown at the bottom of the next page requires to transfer the yielded coroutine into Θ, after where it is from.

CR6, as shown at the bottom of the next page is used when no coroutine has empty receiving part and Pending Type is $\varnothing$, referred as a deadlock state. In such cases, the composition result is E followed by all the remaining coroutines from the original list. Θ is equal to Θ_1 in this case.

2) RESUMING

When Pending Type t is not $\varnothing$, the resume operation, CR7, as shown at the bottom of the next page, is triggered. We find the first coroutine θ that can receive t, and resume it. $\mathbf{match}(\cdot, \cdot)$, defined in Fig. 5, checks if two types can match and returns conditions D. Conditions D are in the form of variable bindings as θ may contain variables. For example $\mathbf{match}(\mathrm{Int}^5, \mathrm{Int}^n) = \{n = 5\}$. If in no way can two types match, $\perp$ is returned. The absorbing element $\perp$ joining with any condition is $\perp$. **match** also handles constraint types by satisfying the constraint p.

CR8, as shown at the bottom of the next page stipulates that if none of the coroutines in Θ can process the Pending Type t, then t gets added to the end of E.

CR9, as shown at the bottom of the next page outlines how a coroutine receives other coroutines. Line 2 in CR9 finds the first coroutine whose head $\mathbf{h}(s)$ is a pattern of coroutine, then Line 3 aims to match this pattern with another coroutine θ', returning conditions D. Then we apply the condition to $[\mathbf{t}(s); u]$, and also remove the received coroutine from the result by using the minus sign. This rule, in combination with CR5, is essential for handling first-class coroutines.

If all coroutines in Θ cannot yield, CR10, as shown at the bottom of the next page loops the types in E. It finds the first type t in E such that one coroutine in Θ can receive t. This rule is critical in composing contracts in REModel because developers may require and yield data in mismatching order. Details are in Section III-C.

III. ANALYZING REMODEL FILES

A. SYNTAX OF THE REMODEL LANGUAGE

Although Yang claims that REModel files are written in Object Constraint Language (OCL) v2.4 [11], there are many keywords or elements foreign to OCL in the example files [18], such as `UseCaseModel` and `@Description`.

The lexer rules of the REModel Language are almost identical with the standard OCL but REModel uses two successive slashes for comment rather than two dashes [11].

For parser rules, we start with contract sections. A contract section is similar to the Operation declaration in OCL. Its abstract structure is crystallized in Fig. 6. In terms of the `type` element, REModel has a notation to write out enumeration types in type ::= ⋯ | **ID** `[` ⟨**ID**, **ID**, ⋯⟩ `]`. One enum type can be found in Listing 1.

As Fig. 7 illustrates, in OCL, an expression element can be either `logicalExpression`, `conditional-Expression`, `lambdaExpression`, or `letExpression`. Then, `factor2Expression` is a component of a `logicalExpression`.

In REModel, `letExpression` is improved to bear multiple variables. OCL allows a collection operation, i.e., `factor2Expression`, to take an iterator, but REModel adopts the Church style [19] for these lambda expressions, meaning that a type is specified for the iterator, as shown in Fig. 8. Moreover, the else part of an if-conditional expression can be left out in REModel.

Since REModel can compare the initial and the current state of objects, it permits a `@pre` tag after an expression, which is defined as basicExpression ::= ⋯ | basicExpression `.` **ID** `@pre`?.

B. RECOGNIZING IDENTIFIERS

Identifiers found in a contract definition can refer to parameters of this contract, fields in this class, and global fields in the system class. Hence, our type system has to read the service definition sections and extract fields under `[TempProperty]`. A snippet of service definitions is shown in Listing 1. The service whose name ends with the word System is the system class, fields of which are available for all services.

Furthermore, we have to know the type inheritance by looking into actor sections. From Listing 2 we know that the type Faculty is a subtype of the type User. Given the relationship, if a coroutine type is [User; Book], a type Faculty should be able to activate this coroutine. The subtype handling is the last step of typing.

C. TYPING CONTRACTS AS COROUTINES

Equipped with the identifiers and type information of a REModel file, we are able to parse the file and give a coroutine type to each contract block defined therein. Basically, the precondition element becomes the receiving part of a coroutine, and the postcondition element becomes the yielding part. We ignore the parameter list and the return type.

Type mapping Γ is built according to Section III-B which maps identifiers to their types. However, if an identifier is a global field or a class field, we use the name of the identifier rather than its type in order to distinguish them from data retrieved from other places.

For the contract `enterItem` in Listing 3, its inferred coroutine type is [CurrentSale, Item; CurrentSale, Item, SalesLineItem, CurrentSaleLine]. This type is calculated by Typing Rule 1 to 6.

$$\frac{\begin{array}{c}\Gamma + \mathbf{m}(e_1) \vdash \mathbf{r}(e_2) = t_1 \\ \Gamma + \mathbf{m}(e_1) \vdash \mathbf{a}(e_3) = t_2 \\ \Gamma + \mathbf{m}(e_1) \vdash \mathbf{d}(e_3) = t_3\end{array}}{\Gamma \vdash \begin{array}{l}\texttt{Contract} \cdots \texttt{\{} \\ \texttt{definition:}\ e_1 \\ \texttt{precondition:}\ e_2 \\ \texttt{postcondition:}\ e_3\texttt{\}}\end{array} : [t_1; t_1 - t_3 + t_2]} \tag{TR1}$$

Typing Rule TR1 stipulates that in order to type a contract block, we need to look into its definition section, precondition and postcondition section. Function **m** extracts type mapping from the definition context. For listing 3, $\mathbf{m}(\texttt{item : Item = Item.allInstance()}\ldots) = \langle \text{item} : \text{Item} \rangle$. Type `Set(t)` in OCL is mapped to t^*.

The receiving part is typed based on the precondition; the yielding part is mainly based on the postcondition.

$$(\varnothing, \cdot) \vdash \mathsf{o}(\Theta) \Rightarrow (\mathbf{h}(s), \cdot) \vdash \mathsf{o}(\langle \Theta_1, [\varnothing; \mathbf{t}(s)], \Theta_2 \rangle) \quad \text{provided}$$
$$(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.\theta = [\varnothing; s] \wedge s \neq \varnothing) \tag{CR4}$$

$$(\varnothing, \cdot) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \cdot) \vdash \mathsf{o}(\langle \Theta_1, [\varnothing; \mathbf{t}(s)], [u; v], \Theta_2 \rangle) \quad \text{provided}$$
$$(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.\theta = [\varnothing; s] \wedge \mathbf{h}(s) = [u; v]) \tag{CR5}$$

$$(\varnothing, E) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \varnothing) \vdash \langle E, \Theta_1 \rangle \quad \text{provided } (\varnothing, \Theta_1, \varnothing) = \mathbf{first}(\Theta, \lambda\theta.\theta = [\varnothing; \cdot]) \tag{CR6}$$

$$(t, \cdot) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \cdot) \vdash \mathsf{o}(\langle \Theta_1, [\mathbf{t}(s); u]\,[D], \Theta_2 \rangle) \quad \text{provided}$$
$$(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.\theta = [s; u] \wedge \mathbf{match}(t, \mathbf{h}(s)) = D) \tag{CR7}$$

$$(t, E) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \langle E, t \rangle) \vdash \mathsf{o}(\Theta) \quad \text{provided}$$
$$(\varnothing, \Theta_1, \varnothing) = \mathbf{first}(\Theta, \lambda\theta.\theta = [s; \cdot] \wedge \mathbf{match}(t, s) = \bot) \tag{CR8}$$

$$(\varnothing, \cdot) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \cdot) \vdash \mathsf{o}(\langle \Theta_1, [\mathbf{t}(s); u]\,[D], \Theta_2 \rangle - \theta') \quad \text{provided}$$
$$(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.\theta = [s; u] \wedge \mathbf{h}(s) = [\cdot; \cdot])$$
$$\text{and } (\theta', \cdot, \cdot) = \mathbf{first}(\Theta - \theta, \lambda\theta'.\mathbf{match}(\theta', \mathbf{h}(s)) = D) \tag{CR9}$$

$$(\varnothing, E) \vdash \mathsf{o}(\Theta) \Rightarrow (\varnothing, \langle E_1, E_2 \rangle) \vdash \mathsf{o}(\langle \Theta_1, [\mathbf{t}(s); u]\,[D], \Theta_2 \rangle) \quad \text{provided}$$
$$(\varnothing, \cdot, \cdot) = \mathbf{first}(\Theta, \lambda\theta.\theta = [\varnothing; \cdot]) \text{ and}$$
$$(t, E_1, E_2) = \mathbf{first}\big(E, \lambda t.(\theta, \Theta_1, \Theta_2) = \mathbf{first}(\Theta, \lambda\theta.\theta = [s; u] \wedge \mathbf{match}(t, \mathbf{h}(s)) = D)\big) \tag{CR10}$$

$$\mathbf{match}([s;t],[u;v]/p)=\begin{cases}D & \text{if } D=\mathbf{match}(s,u)\cup\mathbf{match}(t,v) \text{ and } p(D),\\ \perp & \text{otherwise.}\end{cases}$$
$$\mathbf{match}(s,[\cdot;\cdot])=\perp$$
$$\mathbf{match}(s,t/p)=\begin{cases}D & \text{if } \exists D.\mathbf{h}(t)[D]=s \text{ and } p(D),\\ \perp & \text{otherwise.}\end{cases}$$

FIGURE 5. Definition of match for matching two types.

What's more, the receiving part has to be added to the yielding part unless the postcondition explicitly removes it. The plus (+) operation stands for sequence joining. The minus (−) operation removes elements from the first operand. Removing an absent element has no effect. **r**, **a**, **d** stand for getting required, added, deleted types from given expression. Their definitions can be found in TR4, TR5, TR6, respectively.

Before we proceed to precondition and postcondition sections, we have to normalize logical expressions into conjunctive normal forms, and apply TR2 to individual terms. Also let-expressions must be processed by TR3.

$$\frac{\Gamma\vdash\mathbf{r}(e_1)=t_1 \quad \Gamma\vdash\mathbf{r}(e_2)=t_2}{\Gamma\vdash\mathbf{r}(e_1\wedge e_2)=t_1+t_2}$$
$$\frac{\Gamma\vdash\mathbf{a}(e_1)=t_1 \quad \Gamma\vdash\mathbf{a}(e_2)=t_2}{\Gamma\vdash\mathbf{a}(e_1\wedge e_2)=t_1+t_2}$$
$$\frac{\Gamma\vdash\mathbf{d}(e_1)=t_1 \quad \Gamma\vdash\mathbf{d}(e_2)=t_2}{\Gamma\vdash\mathbf{d}(e_1\wedge e_2)=t_1+t_2} \tag{TR2}$$

$$\frac{\Gamma+\mathbf{m}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\vdash\mathbf{r}(e)}{\Gamma\vdash\mathbf{r}(\texttt{let}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\texttt{in}\ e)}$$
$$\frac{\Gamma+\mathbf{m}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\vdash\mathbf{a}(e)}{\Gamma\vdash\mathbf{a}(\texttt{let}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\texttt{in}\ e)}$$
$$\frac{\Gamma+\mathbf{m}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\vdash\mathbf{d}(e)}{\Gamma\vdash\mathbf{d}(\texttt{let}\langle\mathbf{ID}_1:t_1,\mathbf{ID}_2:t_2,\cdots\rangle\texttt{in}\ e)} \tag{TR3}$$

For precondition, TR4 defines the function **r**, standing for "required". Required types translate to the receiving part of the coroutine type. $\Gamma[e]$ looks up the value of key e. For example, $\langle\text{item}:\text{Item}\rangle[\text{item}]$ = Item. If operation `oclIsUndefined` returns false on an identifier or something must be included in all instances, we know the type of this identifier is required for this contract. Property checks, such as `item.StockNumber > 0` in Listing 3, are ignored.

The conjunction and disjunction operator in Boolean algebra are communicative, but our sequence is not. For instance, the postcondition section in one contract may yield User and Book, while the precondition section in another may require Book and User, the order reversed. That's why CR10 is crucial for composing contracts as it can feed the second element Book of the yielding part to the first element of the receiving part.

```
Service CoCoMESystem {
  [Operation]
  openCashDesk(cashDeskID)
  closeCashDesk(cashDeskID)
  openStore(storeID)
  closeStore(storeID)

  [TempProperty]
  CurrentCashDesk : CashDesk
  CurrentStore : Store
}

Service ProcessSaleService {
  [Operation]
  makeNewSale()
  enterItem(barcode, quantity)

  [TempProperty]
  CurrentSaleLine : SalesLineItem
  CurrentSale : Sale
  CurrentPaymentMethod : PaymentMethod[CASH|CARD]
}
```

LISTING 1. Two service blocks in REModel.

$$\begin{aligned}
&\Gamma\vdash\mathbf{r}(\neg e\texttt{.oclIsUndefined()})=\Gamma[e]\\
&\Gamma\vdash\mathbf{r}(\neg e_1\texttt{.allInstance()->excludes(}e_2\texttt{)})\\
&=\Gamma[e_2]\\
&\Gamma\vdash\mathbf{r}(\neg e)=\varnothing\\
&\Gamma\vdash\mathbf{r}(e_1\texttt{.allInstance()->includes(}e_2\texttt{)})\\
&=\Gamma[e_2]
\end{aligned} \tag{TR4}$$

TR5 is called by TR1 to process the added types for the postcondition section. It checks if a contract modifies a class property or creates an entity instance. In detail, when an object calls `oclIsNew()`, this object is for sure newly created and will be added to an entity manager, such as a database or the block on a blockchain. `oclIsNew()` is an OCL operation designed specifically for the post-condition section [11]. When a class field is assigned, we return the name of the field. The `includes()` and `excludes()` operations are handled the same as TR4.

$$\begin{aligned}
&\Gamma\vdash\mathbf{a}(e.\texttt{oclIsNew()})=\Gamma[e]\\
&\Gamma\vdash\mathbf{a}((\texttt{self.})?\mathbf{ID}=e_2)=\Gamma[\mathbf{ID}]\\
&\Gamma\vdash\mathbf{a}(\neg e_1\texttt{.allInstance()->excludes(}e_2\texttt{)})\\
&=\Gamma[e_2]\\
&\Gamma\vdash\mathbf{a}(\neg e)=\varnothing\\
&\Gamma\vdash\mathbf{a}(e_1\texttt{.allInstance()->includes(}e_2\texttt{)})\\
&=\Gamma[e_2]
\end{aligned} \tag{TR5}$$

contractDefinition ::=
  `Contract` **ID** `::` **ID**
  `(` parameterDeclarations? `)`
  (`:` type)?
  `{` definitions?
    precondition?
    postcondition? `}`

definitions ::=
  `definition:` ⟨definition, definition, ⋯⟩

definition ::=
  ID `:` type `=` factor2Expression

precondition ::=
  `precondition:` expression

postcondition ::=
  `postcondition:` expression

FIGURE 6. The abstract syntax for the contract element.

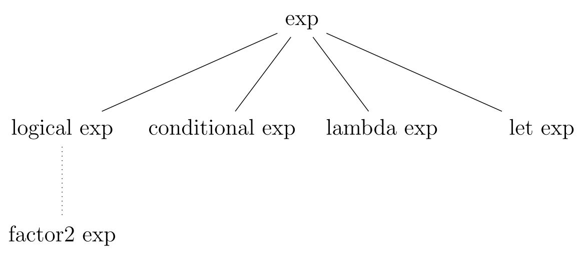


FIGURE 7. Types of expressions in REModel/OCL.

```
Actor User {
  @Description( "The user")
  searchBook
  listBookHistory
  makeReservation
  recommendBook
  cancelReservation
}

Actor Faculty extends User {
  @Description( "The faculty user") }
```

LISTING 2. Actor definitions in REModel.

The postcondition section can contain assertions that an entity must be deleted. These terms in the conjunctive normal form will be matched by TR6. As a side note, based on OCL, the includes and excludes operation can take an expression, but R_{17} and R_{18} in [20] put solely an identifier as the argument of the two operations. Consequently, we assume expressions are not allowed at this position.

$$\Gamma \vdash \mathbf{d}(e\texttt{.oclIsUndefined()}) = \Gamma[e]$$
$$\Gamma \vdash \mathbf{d}(e_1\texttt{.allInstance()->excludes(}e_2\texttt{)}) = \Gamma[e_2] \qquad \text{(TR6)}$$

letExpression ::=
  `let` ⟨**ID** `:` type, **ID** `:` type, ⋯⟩
  `in` expression

factor2Exp ::= ⋯
 | factor2Exp `->any(` identifier `:` type `|` expression `)`
 | factor2Exp `->forAll(` identifier `:` type `|` expression `)`
 | factor2Exp `->select(` identifier `:` type `|` expression `)`

conditionalExpression ::=
  `if` expression
  `then` expression
  (`else` expression)?
  `endif`

FIGURE 8. The syntax for expressions in REModel.

The coroutine type returned by TR1 is not final yet. If any type in the receiving part is a super type, we replace all occurrences of this type in the coroutine type with a variable, and add an upper type bound constraint $<:$ [21]. Retrieving child-parent relationship has been covered in Section III-B.

IV. APPLICATIONS

We created the concrete syntax of REModel by extending the syntax of OCL retrieved from ANTLR's GitHub repository.[1] Our coroutine type system was developed in.NET Core 3.1 (C#). The source code,.remodel files for testing, and test cases are all available on GitHub.[2] The `RequirementAnalysis` folder implements the REModel parser and typing rules (typer), and the `GeneratorCalculation` folder implements composition rules (composer).

Our first experiment is to type four case studies provided by [20] to demonstrate the validity and capability of our system. The Composer component is indeed standalone. Hence, the second experiment is using Composer to model other programming languages. This shows the versatility of our type system.

A. USING YIELDING ORDER

The RM2PT repository [18] provides a number of case studies. The case studies have been used in [20] to validate the RM2PT platform. Since our type system is used with the platform, it is natural to employ the same case studies to validate our invention. The used case studies are Supermarket System (CoCoME), Library Management System (LibraryMS), Automated Teller Machine (ATM), and Loan Processing System (LoanPS); each has a.remodel file.

[1]https://github.com/antlr/grammars-v4/tree/master/ocl
[2]https://github.com/gqqnbig/coroutine-program/

```
Contract ProcessSaleService::enterItem(barcode : Integer,
      quantity : Integer) : Boolean {

  definition:
    item:Item = Item.allInstance()->any(i:Item | i.Barcode
          = barcode)

  precondition:
    CurrentSale.oclIsUndefined() = false and
    CurrentSale.IsComplete = false and
    item.oclIsUndefined() = false and
    item.StockNumber > 0

  postcondition:
    let sli:SalesLineItem in
    sli.oclIsNew() and
    self.CurrentSaleLine = sli and
    sli.BelongedSale = currentSale and
    CurrentSale.ContainedSalesLine->includes(sli) and
    item.StockNumber = item.StockNumber@pre - quantity and
    sli.Subamount = item.Price * quantity and
    SalesLineItem.allInstance()->includes(sli) and
    result = true
}
```

LISTING 3. The prototype of the enterItem function.

We type the four requirement models as coroutines and check whether the inferred coroutine types are correct. Listing 4 is a portion of the typing log when we process the CoCoME model. Line 1 to 11 show the type of each contract. (Some trivial contracts are deleted to save space.) Line 13 is the composed type from the types above, by our composition rules. This composed type means after running these contracts, the system will have set CurrentStore, CurrentCashDesk, Sale, and so on.

In the meantime, since the yielding order during composition tells the time when the prerequisite of a coroutine has met and the coroutine is ready to execute, we hooked the execution of CR1 and CR4. Line 15 to 26 are the execution order of the aforementioned coroutines; namely we start with createStore, then compose with openStore, and all the way to deleteItem. Some contracts appear a couple of times because each yielding is recorded and we usually only need to look at the first occurrence.

The execution order is useful in that firstly, for requirement verification, users can check it against the sequence diagram in a requirement model. If the two orders do not match, this requirement model is contradictory. Secondly, users can use the order to develop integration tests. When we were working on paper [7], we did not have this coroutine type system and had to manually figured out the calling order in their integration tests if the sequence diagram was not complete.

To provide a negative test case, suppose the model author forgot to include CurrentSale in the `enterItem` contract in Listing 3, the type of `enterItem` would be [Item; ⟨Item, SalesLineItem, CurrentSaleLine⟩]. This type would be composed before `makeNewSale`, creating a contrast to the sequence diagram.

B. USING THE COMPOSER ITSELF

Our composition rules are not necessarily bundled with the type system, but can be used by themselves. In this subsection we demonstrate the use of the composer to solve a Prolog query. We borrow a family knowledge base snippet found in [22] and list the rules in Listing 5. Given a query `parent(X,jane), male(X).`, Prolog will tell `X=sam`. Unlike Prolog, our composer cannot return a solution, but it is capable to prove whether a given answer is true or false.

Apparently the REModel type system does not recognize the Prolog syntax, so we assume there is a hypothetical type system that gives a coroutine type for each Prolog rule, and the typing result is given in Fig. 9.

We have $\circ(\Theta) = [\varnothing; \text{Yes}]$. On the other hand, if we replace *answer* to $[\varnothing; \text{Sue}]$, CR6 triggers and $\circ(\Theta)$ deadlocks, so we know Sue is not a solution.

One point of consequence is the presence of constrained types in Fig. 9. They help the composition result independent from the type order in Θ. Specifically, if we remove the constraint from *childOther* and it becomes $[(\text{Child}, x, y); \text{No}]$, and it is placed before *child1*, then *childOther* will compose with *parent* right away, yielding No as a result.

V. DISCUSSION AND LIMITATION

We ran the REModel type system against four case studies of RM2PT. The process did not throw exceptions and we manually verified the typing results because there is not another automation system as ground truth. We also added tests in GitHub Actions to maximize the confidence of correctness. To test the composer, we selected 6 use cases out of 4 case studies where `library.remodel` and `cocome.remodel` each contributed 2 cases. Totally we composed 43 out of 115 contracts defined in the 4 case studies. We did not compose all contracts because a REModel file typically had contract `addXX`, `modifyXX`, `queryXX`, `deleteXX`. Composing these CRUD operations is not interesting.

Apart from the integration tests, there are 26 unit tests on the composition engine. All these tests are verified by GitHub and indicated by a green tick next to each commit.

There are two contracts `inputCard` and `inputPassword` in `atm.remodel` that our type system does not process correctly. The post-condition in `inputCard` is a conditional statement (see Listing 6) that sets CardIDValidated to true or false based on the card validation result. However, we have no typing rules in regard to if-conditions because our coroutine type expressions do not express branching. Consequently, this contract is typed $[\varnothing; \varnothing]$, so it cannot compose with other contracts. Contract `inputPassword` has a similar issue.

Our work has some limitation. As we mentioned, our type system does not handle if-conditions well and we do not have mechanism to refine the condition or add the dependency from a precondition to a postcondition.

Second, our type system only discriminates objects being null or non-null, and value checks such as `item.StockNumber > 0` are ignored. Since a type system only concerns types, a different system or a dependent type system—where type level functions are pervasive—may be needed to understand fine-grained checks.

Last, our type system does not divide coroutine types into groups. If a requirement model contains contracts for both CoCoME and ATM, the type system still composes all types together, and the result may not make sense. When we were typing RM2PT case studies, we had to manually find clean-up contracts (such as *closeStore* and *deleteItem*), and set them to be composed last by using tuples. If we should forget to do so, the composer might compose *closeStore* right after *openStore*, blocking *makeNewSale* and subsequent operations. An upper stream component will be handy to filter and group contracts, before calling the composer.

Coroutines are ideal to present side-effects, i.e., changes to global data or performs IO [23], [24], and do not capture parameters of a function and the return value. Therefore we do not have typing rules reading the parameter list of a contract. Pure functions, such as query operations, are typed to have the identical receiving part and yielding part because pure functions solely require something in memory but do not change it. To model parameters and return values, we should use the traditional function types $f : a \to b$.

VI. RELATED WORK

Model-driven engineering (MDE) can improve implementation productivity. TopCased [25] is a MDE platform providing modeling languages and code generation. It is widely used in Europe. RM2PT [20] implements model-to-text (M2T) transformation [26], and cuts coding time from hours to one second. By using MetaCase, another piece of MDE software, a sports watch company observed productivity increase of 750% [27].

A model in MDE simply refers to an abstraction of data or behavior that captures knowledge, enables automation, and facilitates communication [28]. A file can be a requirement model; the format a requirement model must conform to is also a model, or we say metamodel. In this sense, languages are metamodels [29], [30]. A valid model is said to conform to its metamodel if it has no syntax errors and fulfills other constrains in the metamodel. For a.remodel file, it must at least pass syntax check of the REModel language, and its pre- and post-conditions match the specification of the Interaction section, so on and so forth.

REModel is basically a variation of OCL. OCL is typed, with basic types such as Integer, Boolean, and String, and parameterized sets such as Set(T) and Sequence(T), as well as tuples [11]. The concrete types in our coroutine type system map to the basic types in OCL. Distefano et al. [31] gives a formal semantics to OCL invariants and pre-, post-conditions, and run model checking of object-oriented programs. Nevertheless, their approach does not support

```
openStore: [Store; <Store, CurrentStore>]
openCashDesk: [<CashDesk, CurrentStore>; <CashDesk,
       CurrentStore, CurrentCashDesk>]
makeNewSale: [CurrentCashDesk; <CurrentCashDesk, Sale,
        CurrentSale>]
enterItem: [<CurrentSale, Item>; <CurrentSale, Item,
       SalesLineItem, CurrentSaleLine>]
makeCashPayment: [CurrentSale; CashPayment]
createStore: [Void; Store]
deleteStore: [Store; Void]
createCashDesk: [Void; CashDesk]
deleteCashDesk: [CashDesk; Void]
createItem: [Void; Item]
deleteItem: [Item; Void]

[Void; <CurrentStore, CurrentCashDesk, Sale,
       CashPayment, SalesLineItem, CurrentSaleLine>]

createStore -> openStore ->
openStore -> openStore ->
createCashDesk -> openCashDesk ->
createItem -> openCashDesk ->
openCashDesk -> openCashDesk ->
makeNewSale -> makeNewSale ->
makeNewSale -> makeNewSale ->
enterItem -> enterItem ->
makeCashPayment -> makeCashPayment ->
enterItem -> enterItem ->
enterItem -> deleteStore ->
deleteCashDesk -> deleteItem
```

LISTING 4. Partial typing log of the CoCoME requirement model.

```
child(john,sue). child(john,sam).
child(jane,sue). child(jane,sam).
child(sue,george). child(sue,gina).

male(john). male(sam). male(george).
female(sue). female(jane). female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).

opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).

grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

LISTING 5. A family knowledge base in Prolog.

type inheritance. Reference [32] allows to write temporal properties in OCL, for instance work 2 will start only when work 1 is finished. These properties can be checked by Tina model checker [33]. Reference [4] verifies behavioral properties of a model with Petri Nets and can give a counterexample if a property (assertion) fails. A formal approach is proposed by [34] to define and analyze domain-specific modeling languages (DSML). It represents DSML metamodels and their conforming models as a Maude specification [35]. Fiacre [36] is a formal specification language that targets both the behavioral and timing aspects of real-time systems. Its authors transform UML and OCL to Fiacre, and then perform Tina verification. Overall, prior work mainly studies constraints in a single OCL context. In contrast, our work focuses on the interplay of a set of OCL contexts (named contract in REModel), and can work out temporal properties from pre- and post-conditions by modeling contract blocks as coroutines.

Coroutines are included in assorted modern programming languages. In particular, coroutines in Kotlin can dispatch user actions faster comparing to Java threads [37].

$$
\begin{aligned}
\Theta = \langle \mathit{child1} &: [(\text{Child}, \text{John}, \text{Sue}); \varnothing]\,, \\
\mathit{child2} &: [(\text{Child}, \text{Jane}, \text{Sue}); \varnothing]\,, \\
\mathit{child3} &: [(\text{Child}, \text{Sue}, \text{George}); \varnothing]\,, \\
\mathit{child4} &: [(\text{Child}, \text{John}, \text{Sam}); \varnothing]\,, \\
\mathit{child5} &: [(\text{Child}, \text{Jane}, \text{Sam}); \varnothing]\,, \\
\mathit{child6} &: [(\text{Child}, \text{Sue}, \text{Gina}); \varnothing]\,, \\
\mathit{childOther} &: [(\text{Child}, x, y); \text{No}]\,/(x, y) \notin \\
&\quad \big\{(\text{John}, \text{Sue}), (\text{Jane}, \text{Sue}), (\text{Sue}, \text{George}), \\
&\quad (\text{John}, \text{Sam}), (\text{Jane}, \text{Sam}), (\text{Sue}, \text{Gina})\big\}, \\
\mathit{male1} &: [(\text{Male}, \text{John}); \varnothing]\,, \\
\mathit{male2} &: [(\text{Male}, \text{Sam}); \varnothing]\,, \\
\mathit{male3} &: [(\text{Male}, \text{George}); \varnothing]\,, \\
\mathit{maleOther} &: [(\text{Male}, x); \text{No}]\,/x \notin \{\text{John}, \text{Sam}, \text{George}\}\,, \\
\mathit{parent} &: [(\text{Parent}, y, x); (\text{Child}, x, y)]\,, \\
\mathit{query} &: [x; \langle(\text{Parent}, x, \text{Jane}), (\text{Male}, x), \text{Yes}\rangle]\,, \\
\mathit{answer} &: [\varnothing; \text{Sam}]\rangle
\end{aligned}
$$

FIGURE 9. The coroutine representation of the family knowledge base in Prolog.

Vector [38] is one of the Model-View-Intent libraries that utilize the coroutine backend. Our coroutine type composition rules are to some extend similar to Vector and other coroutine dispatchers and reducers, but we reduce types rather than values, and have limited capability in arithmetic and logical calculations. We do not concern performance either.

A coroutine system can be categorized into three aspects, namely control-transfer mechanisms, whether coroutines are first-class objects, and whether the control can be suspended within nested calls [39]. In REModel, when a contract calls a built-in function, the function cannot stop the whole contract; hence REModel is a stackless coroutine language. Overall, our type composition rules support asymmetric, first-class, stackless coroutines. Kotlin Coroutines are stackless as well because not every function has access to the CoroutineScope of the parent coroutine. Ikebuchi et al. [40] proposes a high-level language for defining coroutines and the language can be compiled to low-level C code. This approach is useful for implementing security-critical network protocols. Nevertheless, this paper concentrates on the execution logic of a coroutine rather than an overall type of a coroutine, not mention composing. The coroutine type in [41] has three elements, parameters P, return type R, and yielded values Y. It does not include received values because the resume site creates a new instance every time and pass the received values as parameters.

Our type system can be perceived as an integration platform and Górski [42] proposed 6 views for such system, including Integrated Services view, and Contracts view. Our Fig. 2 is indeed the Integrated services view, where each rectangle box is Górski's contract. His Contracts view

```
postcondition:
  if bc.oclIsUndefined() = false
  then
    self.CardIDValidated = true and
    self.InputCard = bc and
  else
    self.CardIDValidated = false and
  endif
```

LISTING 6. Conditional expressions are not typed.

illustrates the cooperation of components, and each contract has a provided interface (a ball) and a required interface (a socket). Górski's contract is dissimilar to the contract keyword in REModel. In REModel, a contract keyword denotes a function, of which the function body is fully specified by pre-conditions and post-conditions. A coroutine with void receiving part means it does not receive data; a coroutine with void yielding part means it does not yield data. Therefore we adopt $\varnothing$, the void type in type theory, in our type system to present a type with no inhabitants [43], [44]. Besides validating requirement models, a type system can check correctness of format string [45], regular expressions [46], and so on.

VII. CONCLUSION

In software engineering, requirements validation is the process of checking whether requirements meet the customers' real needs. It is critically important because requirements errors will lead to extensive rework if those problems are discovered during the later phases of the software development. To verify a requirement file, RM2PT implements a transformation from requirement files to executable prototypes so that developers can verify the requirement by running the prototype. However, it does not have powerful verification tools for models themselves. To compensate for the shortcoming, this paper takes another approach. Rather than transforming a requirement file, we directly verify the file by a formal method to ensure consistency. The file is written in the REModel language, which is a variation of OCL.

We contribute a new type representation and computation method for combining asymmetrical, first-class, stackless coroutines. By formally defining the syntax of REModel, our type system checks each contract section in REModel and infers a coroutine type based on pre- and post-conditions. Then a selection of coroutine types can be combined into a single coroutine type to model the final result of a sequence of operations. The composition operation is expected to produce the same sequence diagram embedded in the requirement model. If not, discrepancies have been spotted.

We evaluated our approach on the four official case studies of RM2PT, namely ATM, CoCoME, LibraryMS, and LoanPS, and confirmed that the coroutine composition results are correct. Moreover, with constrained types and the composition rules, coroutine types can model other programming languages, such as Prolog. The types are composed in a way independent from the activation order. As a result, no matter the clause order of a logical expression in pre-condition or post-condition, contracts are composed to the same type. Subtypes are supported as well.

There are some future work. Firstly, we plan to revise the system to handle fine-grained checks, such as conditional expressions. Secondly, an extra upper stream component can do its part to divide coroutines into groups, or assign priorities to coroutines or contract. Finally, we hope to adapt our coroutine type system for a wider range of requirement models, not only for RM2PT.

REFERENCES

[1] F. Zalila, "Methods and tools for the integration of formal verification in domain-specific languages," Ph.D. dissertation, Inst. Nat. Polytechn. De Toulouse, Toulouse, France, 2014.

[2] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *Proc. ACM/IEEE 20th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS)*, Sep. 2017, pp. 229–236, doi: 10.1109/MODELS.2017.5.

[3] H. F. Hofmann and F. Lehner, "Requirements engineering as a success factor in software projects," *IEEE Softw.*, vol. 18, no. 4, pp. 58–66, Jul. 2001, doi: 10.1109/MS.2001.936219.

[4] F. Zalila, X. Crégut, and M. Pantel, "Leveraging formal verification tools for DSML users: A process modeling case study," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*. Berlin, Germany: Springer, 2012, pp. 329–343, doi: 10.1007/978-3-642-34032-1_34.

[5] G. Atladottir, E. T. Hvannberg, and S. Gunnarsdottir, "Comparing task practicing and prototype fidelities when applying scenario acting to elicit requirements," *Requirements Eng.*, vol. 17, no. 3, pp. 157–170, Sep. 2012, doi: 10.1007/s00766-011-0131-2.

[6] Y. Yang, X. Li, W. Ke, and Z. Liu, "Automated prototype generation from formal requirements model," *IEEE Trans. Rel.*, vol. 69, no. 2, pp. 632–656, Jun. 2020, doi: 10.1109/TR.2019.2934348.

[7] Q. Gu, W. Ke, and Y. Yang, "Transformation from MVC applications to smart contracts," in *Proc. IEEE 20th Int. Conf. Embedded Ubiquitous Comput. (EUC)*, Dec. 2022, pp. 104–111, doi: 10.1109/EUC57774.2022.00025.

[8] T. Bao, J. Yang, Y. Yang, and Y. Yin, "RM2Doc: A tool for automatic generation of requirements documents from requirements models," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, May 2022, pp. 188–192, doi: 10.1145/3510454.3516850.

[9] Y. Yang, Y. Liu, T. Bao, W. Wang, N. Niu, and Y. Yin, "DeepOCL: A deep neural network for object constraint language generation from unrestricted nature language," *CAAI Trans. Intell. Technol.*, 2023, doi: 10.1049/cit2.12207.

[10] S. Reyal, S. E. R. Siriwardana, A. Kugathasan, S. Godage, D. Nissanka, S. Kalindu, and P. Uduwana, "An investigation into UI generation compliant with HCI standards ensuring artifact consistency across SDLC," in *Proc. 21st Int. Conf. Adv. ICT Emerg. Regions (ICter)*, Dec. 2021, pp. 93–98, doi: 10.1109/ICter53630.2021.9774787.

[11] (2014). *Object Constraint Language*, Object Management Group Standard 2.4. [Online]. Available: https://www.omg.org/spec/OCL/2.4/About-OCL

[12] M. Ozkaya and F. Erata, "A survey on the practical use of UML for different software architecture viewpoints," *Inf. Softw. Technol.*, vol. 121, May 2020, Art. no. 106275, doi: 10.1016/j.infsof.2020.106275.

[13] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. NC, USA: The Pragmatic Bookshelf, 2013.

[14] K.-F. Faxén, "A static semantics for Haskell," *J. Funct. Program.*, vol. 12, nos. 4–5, pp. 295–357, Jul. 2002, doi: 10.1017/s0956796802004380.

[15] S. Chang, A. Knauth, and B. Greenman, "Type systems as macros," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Lang.*, Jan. 2017, pp. 694–705, doi: 10.1145/3009837.3009886.

[16] A. Bove, P. Dybjer, and U. Norell, "A brief overview of Agda—A functional language with dependent types," in *Theorem Proving in Higher Order Logics*. Berlin, Germany: Springer, 2009.

[17] M. P. Papazoglou, P. Georgiadis, and D. G. Maritsas, "An outline of the programming language simula," *Comput. Lang.*, vol. 9, no. 2, pp. 107–131, 1984, doi: 10.1016/0096-0551(84)90018-3.

[18] *RM2PT Case Studies*. Accessed: Jan. 12, 2024. [Online]. Available: https://github.com/RM2PT/CaseStudies

[19] A. Bove and P. Dybjer, "Dependent types at work," in *Language Engineering and Rigorous Software Development*. Berlin, Germany: Springer, 2008, pp. 57–99, doi: 10.1007/978-3-642-03153-3_2.

[20] Y. Yang, X. Li, Z. Liu, and W. Ke, "RM2PT: A tool for automated prototype generation from requirements model," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 59–62, doi: 10.1109/ICSE-Companion.2019.00038.

[21] V. Layka and D. Pollak, "Scala type system," in *Beginning Scala*. CA, USA: Springer, 2015, pp. 133–151, doi: 10.1007/978-1-4842-0232-6_8.

[22] H. J. Levesque, *Thinking as Computation: A First Course*. Cambridge, MA, USA: MIT Press, 2012.

[23] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, Jul. 1963, doi: 10.1145/366663.366704.

[24] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, vol. 1. Reading, MA, USA: Addison-Wesley, 1997, doi: 10.1137/1011017.

[25] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. D. Zilio, M. Filali, and F. Vernadat, "Formal verification of AADL specifications in the Topcased environment," in *Proc. Int. Conf. Reliable Softw. Technol.* Cham, Switzerland: Springer, 2009, pp. 207–221, doi: 10.1007/978-3-642-01924-1_15.

[26] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool, 2017, doi: 10.1016/j.procs.2015.08.383.

[27] J. Kärnä, J.-P. Tolvanen, and S. Kelly, "Evaluating the use of domain-specific modeling in practice," in *Proc. 9th OOPSLA Workshop Domain-Specific Model.*, 2009, pp. 1–7.

[28] E. Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. IN, USA: Dog Ear Publishing, 2014.

[29] R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "A tutorial on metamodelling for grammar researchers," *Sci. Comput. Program.*, vol. 96, pp. 396–416, Dec. 2014, doi: 10.1016/j.scico.2014.05.007.

[30] R. F. Paige, N. Matragkas, and L. M. Rose, "Evolving models in model-driven engineering: State-of-the-art and future challenges," *J. Syst. Softw.*, vol. 111, pp. 272–280, Jan. 2016, doi: 10.1016/j.jss.2015.08.047.

[31] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," in *Proc. ECOOP Workshop Defining Precise Semantics UML*, 2000, pp. 1–10.

[32] B. Combemale, X. Cregut, P.-L. Garoche, X. Thirioux, and F. Vernadat, "A property-driven approach to formal verification of process models," in *Enterprise Information Systems*. Funchal, Madeira: Springer, 2008, pp. 286–300, doi: 10.1007/978-3-540-88710-2_23.

[33] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA—Construction of abstract state spaces for Petri nets and time Petri nets," *Int. J. Prod. Res.*, vol. 42, no. 14, pp. 2741–2756, Jul. 2004, doi: 10.1080/00207540412331312688.

[34] V. Rusu, "Embedding domain-specific modelling languages in Maude specifications," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, Jan. 2011, doi: 10.1145/1921532.1921557.

[35] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, *All About Maude—A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, vol. 4350. Berlin, Germany: Springer, 2007, doi: 10.1007/978-3-540-71999-1.

[36] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat, "Fiacre: An intermediate language for model verification in the topcased environment," in *Proc. 4th Eur. Congr. ERTS Embedded Real Time Softw.*, 2008, p. 8.

[37] K. Chauhan, S. Kumar, D. Sethia, and M. N. Alam, "Performance analysis of Kotlin coroutines on Android in a model-view-intent architecture pattern," in *Proc. 2nd Int. Conf. Emerg. Technol. (INCET)*, May 2021, pp. 1–6, doi: 10.1109/INCET51464.2021.9456197.

[38] K. Chauhan. *Kotlin Coroutines Based MVI Architecture Library for Android*. Accessed: Jan. 12, 2024. [Online]. Available: https://github.com/haroldadmin/Vector

[39] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 2, pp. 1–31, Feb. 2009, doi: 10.1145/1462166.1462167.

[40] M. Ikebuchi, A. Erbsen, and A. Chlipala, "Certifying derivation of state machines from coroutines," *Proc. ACM Program. Lang.*, vol. 6, pp. 1–31, Jan. 2022, doi: 10.5281/zenodo.5628699.

[41] A. Prokopec and F. Liu, "Theory and practice of coroutines with snapshots," in *Proc. 32nd Eur. Conf. Object-Oriented Program.*, 2018, pp. 1–32, doi: 10.4230/LIPIcs.ECOOP.2018.3.

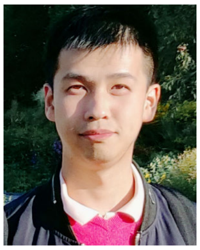
[42] T. Górski, "The 1+5 architectural views model in designing blockchain and IT system integration solutions," *Symmetry*, vol. 13, no. 11, p. 2000, Oct. 2021, doi: 10.3390/sym13112000.

[43] R. L. Constable, "Experience using type theory as a foundation for computer science," in *Proc. 10th Annu. IEEE Symp. Log. Comput. Sci.* Cham, Switzerland: Springer, 1991, pp. 226–243, doi: 10.1109/lics.1995.523262.

[44] U. Norell, *Towards a Practical Programming Language Based on Dependent Type Theory*, vol. 32. Gotaland, Sweden: Chalmers Univ. of Technology, 2007.

[45] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst, "A type system for format strings," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 127–137, doi: 10.1145/2610384.2610417.

[46] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *Proc. 14th Workshop Formal Techn. Java-Like Programs*, Jun. 2012, pp. 20–26, doi: 10.1145/2318202.2318207.



QIQI GU received the M.Sc. degree in computer science from the University of California, Los Angeles, in 2015. He is currently pursuing the Ph.D. degree in computer science with Macao Polytechnic University, Macau, China. He was a Software Development Engineer in Irvine, CA, USA. His research interests include software engineering, especially model-driven engineering and blockchain applications.

WEI KE received the Ph.D. degree from the School of Computer Science and Engineering, Beihang University. He is currently an Associate Professor with the Computing Program, Macao Polytechnic University. His research interests include programming languages, image processing, computer graphics, and component-based engineering and systems. His research interests include the design and implementation of open platforms for applications of computer graphics and pattern recognition.