UML Behavioral Refactoring for the Specification of Complex Software Systems

M.T. Chitra[1,2] and Sherly Elizabeth[2]

[1]University of Kerala
[2]Indian Institute of Information Technology and Management - Kerala
{chitra,sherly}@iiitmk.ac.in

Abstract. Behavioral models play a prominent role in specifying software systems by facilitating the abstract behavior views and analyzing the elementary aspects of a system. The sequence diagram, one of the key behavioral diagrams in UML, provides intuitive ways to capture requirements and scenarios as a sequence of events. The paper proposes a generic framework for effective code generation from UML models. The proposed framework acts as an interexchange format that helps to combine the structural and behavioral constraints of the system objects associated thereby facilitating consistent source code generation. Model Refactoring contributes to improve the software quality and productivity thus mitigating flaws in the system during the design phase itself. The case study presents the refactoring of online simulation of a mill optimization problem in a thermal power plant system.

Keywords: Behavioral diagrams, Model Refactoring.

1 Introduction

The increasing complexity of the software systems demands the advent of novel effective development approaches that can overcome the major drawbacks of the existing development technologies. Model Driven Engineering (MDE), fostered by Object Management Group (OMG) helps reducing the development costs of complex software systems through the use of technologies like Model Driven Architecture (MDA) that supports rigorous analysis of software models [2, 3]. In MDA, models are the primary entities for entitling the system at different levels of abstraction. Models can help to detect inconsistencies or incompleteness in requirement model by evaluating the simulation of the scenarios.

The Unified Modeling Language (UML) is one of the prevalent languages used for modeling complex safety-intensive software systems. It provides a collection of modeling notations for design specifications to build models that describe the different views on a system during various stages such as requirements analysis and design artifacts of software systems. UML Sequence diagrams are now

becoming familiar to represent the behavioral specifications of the complex systems. Sequence diagrams emphasize on modeling the interactions between the collaborating objects participating in the interactions as a time-ordered set of messages [1, 3].

However, UML diagrams fail to specify the semantics in representing the entire information to determine the complete system behavior. The lack of formal semantic specification for UML makes it difficult to analyze the consistency notion in these diagrams. Hence, several transformation approaches are being adopted for analyzing and specifying the consistency of behavioral models which transform the UML models into some semantic domain where the consistency constraints can be represented and validated. The formal specification languages like OCL, TOCL, Z, etc. helps to add additional information to the diagram thereby ensuring the completeness of the model.

OCL is a declarative, side-effect free, formal specification language used along with UML diagrams for specifying the object constraints and queries on the UML models. It precisely defines the well-formedness rules for UML as well as the OMG-related metamodels. OCL is mainly used to specify the invariants of objects as well as the pre and post conditions of the operations [6].

Representing the dynamic behavior of complex time-safety critical systems is a rigorous task as well as a tough research problem to be taken care of. It involves the careful analysis of the semantic aspects with respect to the system constraints. Moreover, there are no model-based techniques or tools available so far for analyzing such temporal properties in UML behavioral diagrams, especially in sequence diagrams. Existing approaches use model transformation techniques that transform the UML models to some other language that supports automated analysis, which are also complex and erroneous.

The Temporal OCL (TOCL) finds its significance in this context. The time and safety related constraints as well as the behavioral specifications, which are hard to express using OCL can be specified using TOCL, a temporal logic extension of OCL. In this paper, we evaluate the impact of OCL and TOCL along with UML behavioral diagrams to completely represent the system behavior especially in safety-critical software environments.

Towards this goal, we provide a refactoring approach to embed the static as well as the temporal constraints involved in the system behavior through a development chain. It constitutes the generation of the behavioral design model using UML sequence diagrams, specification of the constraints using OCL and TOCL, generation of code from the behavioral models by applying the refactoring approach and the execution and analysis of generated code. This approach helps in predicting the system behavior at modeling level itself by comparing the simulation results with real time results. This consequently allows the scientists or the researchers to work exclusively at modeling level in order to obtain the optimized system models. They can analyze the scenario specifications directly from the execution model and can add changes accordingly.

The work focuses on interaction modeling, showcasing the dynamic aspects of the interaction between the participating objects. The dynamic modeling

includes visual specification of the system functionalities in detail, where the functionalities are realized through the message passing between the objects participating in the interaction. The paper discusses about the development of a framework which shows the automated refactoring of UML interaction diagrams, especially the sequence diagram designs, to improve the understandability and maintainability of the design for the efficient source code generation process.

This paper is an extended work and its main new contribution is that the framework has been enhanced with the potential of including a set of safety and temporal constraints in addition to the static constraints of the system [16]. The work proposes a novel approach that yields an interexchange framework to include the complex system behaviors into UML Sequence diagram design as constraints, thereby enriching the model elements with the necessary details of the system without affecting its external behavior.

Incompleteness in the generated source code may often arise due to the absence or failure in representing all possible information regarding the objects that participate in that system. The constraint specification languages play an intelligent role here, by removing this inconsistency in the model design which in turn reflects in the completeness of the generated source code. The proposed framework paves the way for improving the software code quality and productivity fulfilling all the specified system requirements during the code generation process.

The main contributions of this work can be summarized as follows:

- The behavioral refactoring approach in sequence diagram designs is proposed, which paves a way to incorporate static, temporal and safety related constraints into the design and derive a refactored interexchange model from the existing SD specification.
- To provide a generic behavioral pattern for implementing the proposed refactoring methodology in UML models that provides a space for consistent code generation of software systems.

The remainder of this paper is structured as follows: Section 2 gives the related works and identifies and explains the different refactoring activities. Section 3 describes the proposed refactoring approach. Section 4 presents the implementation of the approach through a case study discussion. Finally, Section 5 shows the results and discussions part of the case study and Section 6 concludes.

1.1 Background

The software refactoring is an emerging area where a lot of researches are being carried out on exploring the ways to address refactoring in a consistent manner. Refactoring revolutionizes the design by applying some effective process for improving code quality. The term refactoring was instigated in 1992 by William F. Opdyke in his research work in the context of object-oriented software to support software evolution and reuse. He defined refactoring as “behavior preserving program restructurings or transformations containing particular preconditions that

must be verified before the transformation can be applied in order to make the design of a program clearer and to make it easier to add new features" [7, 8].

Refactorings can also be applied to reduce or eliminate redundant parts of program codes [7]. Martin Fowler defined the process of refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". He proposed the refactoring catalog which focuses on manual refactoring, demonstrated with examples regarding the principles of refactoring, the useful ways to identify and find the associated low-level refactoring(s) that helps fixing a code problem in a controlled and efficient manner [9].

Mens *et al.*, in their review on model-driven refactoring discussed on the latest approaches in model refactoring and also the various challenges encountered while applying refactoring on model level [13]. Mohamed *et al.* discussed on their the existing model refactoring approaces based on feature based model-driven taxonomy [14]. Most of the investigations focus on UML class diagrams, for applying model refactorings [10]. There are only a few approaches that focus refactoring behavioral diagrams and prove their behavior preservation properties in a standard way. G. Sunye *et al.* were the first to present a set of refactoring(s), particularly on UML class models and state machine models, and explained how they can be modeled so as to preserve the behavior of a UML model. They showed that refactorings can be defined for UML in such a way that their behavior-preserving properties are protected, based on OCL constraints defined at the metamodeling level [11].

Alessandro Folli and Tom Mens used Algebraic Graph Grammar tool to support refactoring on the UML designs. They focus on class models and state machine models and define a metamodel similar to the UML meta model as a type graph. The limitation of this work is that this type graph can represent only the simplest version of the UML metamodel [17]. France *et al.* described a meta modeling approach to pattern-based model refactoring in which refactoring(s) are used to introduce a new design pattern instance to the model [18].

Several formalisms have been suggested to examine model refactoring(s). Most of these propose exhibiting model refactoring in a declarative way. Graph transformation theory was used in several works for describing model refactoring and formal properties have been used to review these refactoring(s) [15].

Most of the refactoring approaches focused on code-level refactoring and only a limited works address the model refactoring approaches and especially with class diagrams [13]. But while taking care of the behavior preservation of design models, we need to rely on behavioral models and the constraints associated with them. Only a few works are available in the literature with respect to refactoring of behavioral models. None of the existing approaches can be used to verify operation-based model refactoring that involves changes to operation specification. The core idea presented in this paper focuses on facilitating the integration of system behavioral properties throughout the time point at different abstraction levels one wants to guarantee for the safe or normal execution of the system.

2 The Conceptual Approach

2.1 Refactoring the Behavioral Specifications

Model transformation is the process of modifying the source model to produce the target model. Refactoring is a model transformation approach which restructures the system by altering its internal behavior without affecting the external behavior [1, 2]. Modifying the UML model by enriching with the necessary constraints will significantly increase the quality of the design as well as the generated source code from it. The process of refactoring enables the model to adapt to future extensions.

By definition, refactoring should ensure behavior-preserving transformations of an application. It means that the external behavior of the model before and after the refactoring must remain the same. The major problem faced by designers is to measure the actual impact of modifications on the various design views, as well as on the implementation code which serves as a valid proof to prove the correctness of system and hence showing the behavior preservation property of refactoring approach. Another crucial task here is identifying or determining the exact element to which the refactoring has to be applied.

We are applying the refactoring approach in UML sequence diagrams, which primarily shows the behavioral interactions between the objects participating in the system. Such behavioral transformations need extra care as their change raises some difficulties.

In the proposed work, the model refactoring is applied to the lifeline model element of the sequence diagram design for ensuring the correctness of the system. Since the primary focus of the sequence diagram is to specify the interaction between the collaborating objects, represented as lifelines, it has been chosen as the focal point for refactoring.

We also need to justify why the behavior preservation condition holds for these model transformations:

– Adding the constraint schema into the lifeline model element does not make any modification to the behavior of that element. Rather, we are only giving space to group all the data related to that model element into a common point in order to improve the understandability.

The overall architecture of the refactoring process proposed in this work is shown in Figure 1.

The XML Metadata Interchange (XMI) is OMG standard interchange format used in UML models for exchanging metadata information via XML. It includes information about elements in a model and their relationships. The transformation of the models into XMI is indeed a breakthrough in facilitating interoperability across various tools and platforms [5]. The XMI generated would then be fed into an XMI parser to retrieve the meta model information which inturn is used for the generation of source code. Besides, OCL and TemporalOCL also supplements more precise information specification in the UML behavioral model. The work is an attempt to allow the specification of inter-object scenarios for object-oriented system models in a succinct expressive manner.

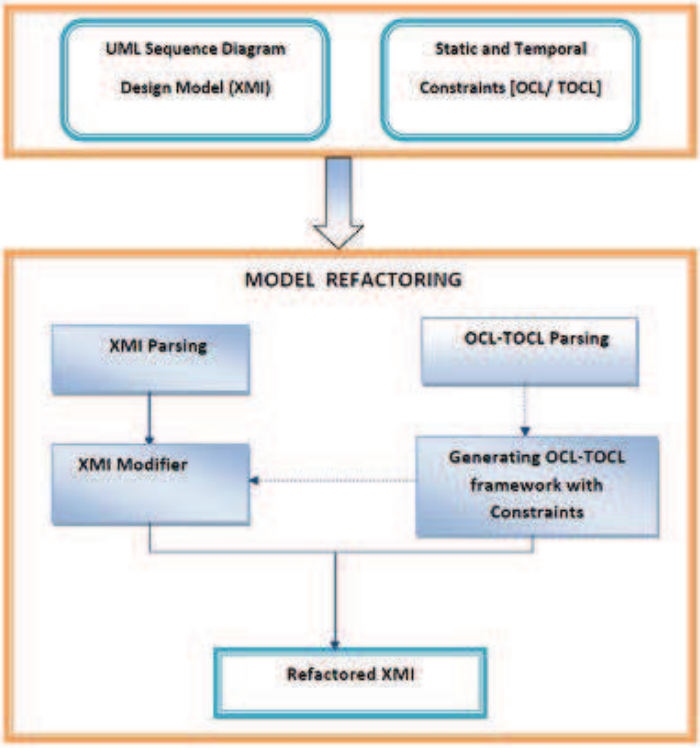

Fig. 1. Architecture of the Refactoring process

2.2 Generation of the OCL-TOCL Framework

The work focuses on developing X-CodeFrame, an XML code framework for representing OCL and TOCL constraints into UML behavioral models, especially the Sequence diagram model thereby facilitating automatic behavioral code generation. This approach helps in simplifying the transformation of the static, temporal and safety-related constraints of the system, thereby smoothening the code generation process.

The constraints that are typically hard to express in OCL, which entitles the temporal and safety related issues of the system are specified using the temporal OCL (TOCL). The static as well as the temporal constraints specified for a system using OCL and TOCL are parsed for framing the constrained elements in order to generate the framework as per the defined schema [schema No-Fig.]. The extracted constraints are then set appropriately into the OCL-TOCL frameworks by matching the context $< classname >$ attribute.

An XML Schema Definition (XSD) typically called an XML schema, is the de facto standard for describing XML documents, controlled by the World Wide Web Consortium (W3C). The XML schema formally defines the XML document

structure accompanying the rules for data content and semantics. The OCL schema proposed here defines an XSD file that provides the design pattern of how the constraints will set within the XMI file obtained after the refactoring process.

The OCL-TOCL schema is the backbone of the proposed UML refactoring process. Based on the defined schema rules, the constraints are embedded into the XMI file of the UML model thereby refactoring it for quality source code generation. The basic types that we are considering for the schema framework generation are the invariants, methods, pre-condition and post-condition operations.

The schema defines the specification rules for framing the constraints appropriately associated with each of the objects participating in the system interaction. For each class defined in the constraint files (OCL/TOCL) there will be an $< extendedConstraints >$ tag generated which is the root element for the constraint construction framework. All the constraints that belong to a particular class will come to an individual $< extendedConstraints >$ tag created for that class. It includes the class name details using the attribute context and also stores the file type within the type attribute.

```
<extendedConstraints context="class_name1" type="ocl/tocl">
  ..........
</extendedConstraints>
```

Fig. 2. Extended constraints

The schema template structure for adding invariants into the OCL-TOCL framework is as follows. The $< MethodDetails >$ tag is the main tag element inside the $< extendedConstraints >$ tag which represents the whole constraint block within a particular class based on the nature of the constraints. The $< body >$ tag element sets the constraints within the $< MethodDetails >$ appropriately. The $< constraints >$ tag is the complex type tag set that helps in identifying all the constraint expressions, operations etc in the $< MethodDetails >$ and include these entities as individual elements under the $< constraint >$ tag within it. The schema template structure for generation of an operation or method is as shown in Fig. 3.

For the execution of a method in any complex system generally includes a precondition which must be considered for a safe system functioning, a body part which includes the actual functionality to be performed and a postcondition operation in which the status of the generated output is checked for further processing of the system. Hence the syntax for generation of the method schema also include these as the key factors while framing data with respect to a method. The $< precondition >$ tag element helps in representing all those conditions for checking the pre-conditions that should satisfy or perform if any before performing the actual method body. The $< body >$ tag include the ex-

```
<extendedConstraints context="class_name1" type="ocl/tocl">
  <MethodDetails name="inv_name1 " returntype=" " type="inv">
    <body>
      <constraints>
        <constraint>
          <condition exprn="expression1"/>
          <condition exprn="expression2"/>
              ..............
        </constraint>
      </constraints>
    </body>
  </MethodDetails>
  <MethodDetails name="inv_name2 " returntype=" " type="inv">
    <body>
      <constraints>
        <constraint>
          <condition exprn="expression1"/>
          <condition exprn="expression2"/>
              ..............
        </constraint>
      </constraints>
    </body>
  </MethodDetails>
      .
      .
      .
</extendedConstraints>
```

Fig. 3. Schema template structure for generation of an operation or method

pression and operations that have to be taken place within that method and the $< postcondition >$ tag represents all those constraints that must have to be executed after the method execution.

The generation of java class files corresponding to the OCL schema is done using JAXB (Java Architecture for Xml Binding), which directly binds the XSD/XML element to particular fields of java classes and vice versa using the properties marshalling and unmarshalling.

The OCL files containing the static constraints and the TOCL files which contain the temporal and safety related constraints for the system are parsed. The constraints are parsed and extracted in a way that helps in fixing the context to which the respective specification refers to in the generated OCL-TOCL frameworks. The constraints extracted are set with the associated java objects of the java classes created using JAXB.

The generated OCL TOCL frameworks are then injected into the XMI of the Sequence diagram model based on the lifelines associated with it.

The OCL -TOCL framework generation process is shown as Algorithm 1.

Algorithm 2 explains the process of filling the constraint objects with appropriate constraint body tag elements.

Algorithm 1. X-CodeFrame Generation

Require: OCL file for sd 's (OCLsd) and also TOCL file for sd 's (TOCLsd), if any.
Ensure: XMLframework of the constraints w.r.t. different contexts referred.

```
1: begin
2: Initialize the ocl-tocl parser with the input
3: Read OCL / TOCL file
4: for each line in file do do
5:    segregate the contents against the context
6:    for each new content in context do do
7:       create an object of ExtendedContraints with type OCL/TOCL
8:       store the object against context
9:       parse the context string
10:      extract the details context data elements like class name, constraint
         type,constraint name, return type if any
11:   end for
12:   for each new constraint within a context do
13:      get the ExtendedConstraints object against a context
14:      add an object of MethodDetails under ExtendedConstraints
15:      invoke populateConstraintBody()
16:   end for
17:   iterate the list of ExtendedConstraints of all context
18:   marshall the objects to the XML content
19:   store it against the context
20: end for
```

2.3 Applying Refactoring to the Model: An Algorithmic approach

Identification and extraction of the exact information that represent functionalities or behavior from the UML models are crucial. This work is proposing a automated refactoring approach for the scenario based UML designs, especially, the UML sequence diagrams, to model the complex system behaviors, henceforth building the behavioral scenarios for individual subcomponents of the system. Refactoring supports a highly dynamic software lifecycle by improving the internal structure of a piece of code block without altering its external behavior. The operation specifications which cannot be directly included in the UML designs are expressed using the static as well as the temporal constraint languages.

Given a model M which consists of model elements which are associated with it to perform the behavioral aspects as per the system requirement specification based on the metamodelling standards (IM). Applying model refactoring in the model $MR = (pre, TR)$, where pre is the precondition or set of rules that model must preserve and satisfy, and TR is the model transformation that is applied to the model.

The Algorithm 3 describes the whole process taken place during the UML refactoring approach.

We illustrate how the refactoring process will happen through the simulation of the mathematical modeling process for optimizing the time factor in a pulverizing process in a thermal power plant system.

Algorithm 2. The populateConstraintBody()

Require: ExtendedConstraints object for a *context*
Ensure: Modified **ExtendedConstraints** object with constraint body
1: begin
2: **if** Precondition **then**
3: create an object of ExtendedConstraints.MethodDetails.Preconditions
4: create Constraints object within Preconditions
5: set the expression string to the Precondition constraints
6: **else if** Postcondition **then**
7: create an object of ExtendedConstraints.MethodDetails.Preconditions
8: create Constraints object within Preconditions
9: set the expression string to the Precondition constraints
10: **else if** Body or inv **then**
11: create an object of ExtendedConstraints.MethodDetails.Preconditions
12: create Constraints object within Preconditions
13: set the expression string to the Precondition constraints
14: **end if**
15: return the modified object of ExtendedConstraints
16: end

Algorithm 3. The Automated UML Refactoring Process

Require: ExtendedConstraints object for a *context*
Ensure: Modified **ExtendedConstraints** object with constraint body
1: begin
2: **if** Precondition **then**
3: create an object of ExtendedConstraints.MethodDetails.Preconditions
4: create Constraints object within Preconditions
5: set the expression string to the Precondition constraints
6: **else if** Postcondition **then**
7: create an object of ExtendedConstraints.MethodDetails.Preconditions
8: create Constraints object within Preconditions
9: set the expression string to the Precondition constraints
10: **else if** Body or inv **then**
11: create an object of ExtendedConstraints.MethodDetails.Preconditions
12: create Constraints object within Preconditions
13: set the expression string to the Precondition constraints
14: **end if**
15: return the modified object of ExtendedConstraints
16: end

3 Case Study : The Coal Pulveriser Optimization Problem

We demonstrate the capability of the proposed refactoring approach though the online coal pulverizing mill optimization problem [16]. In this system, we have considered both the static as well as the behavioral properties involved in the pulverizing process to demonstrate the applicability of the approach. The

main function of the pulverizing mill is to grind and dry the moisturized raw coal supplied to it from the coal storages. The two main classes involved in the pulverisation process in a coal mill are the CoalStorage and the Pulveriser.

The proposed system behavior is modeled using the UML sequence diagram. The safety and time related constraints as well as the static constraints are expressed as TOCL and OCL files respectively. The model is exported as an XMI file and is parsed appropriately for extracting the relevant tag data elements for the code generation process. The OCL as well as the TOCL files are simultaneously parsed to extract the relevant details into the corresponding tag elements in the XMI file using the proposed X-CodeFrame framework.

The UML together with the OCL and TOCL helps in representing the facts that belong to the behavioral level completeness of the system. The constraints of the pulveriser which cannot be typified visually are represented using OCL and TOCL files. The pre and post conditions to be satisfied and the invariants of the system model specified using the OCL file are embedded to the design model by the applying the refactoring approach proposed in the work in order to accomplish the model consistency. The pulveriser optimization process simulation is performed by transforming the mathematical system model using the Object Constraint Language. Along with the sequence diagram model information the temporal as well as the static constraints related with the CoalStorage and the Pulveriser sub systems are also supplied to the model in order to enrich the design data for the code generation process.

3.1 Simulating the Online Coal Pulverisation Process Optimization

The following mathematical model explains the coal pulverisation process [19, 20]. The model is converted to discrete time form for the purpose of online implementation. Figure 2 illustrates the overall online pulverising mill optimization process. The mill model variables are monitored dynamically in real time. The unknown parameters in the equations are estimated using evolutionary computation technique (Genetic Algorithms) and system simulation techniques based on the on-site measurement data. The normal pulverisation process is described mathematically using the following equations:

$$W_{air}(t) = 10\sqrt{\triangle P_{pa}(t)\frac{273}{273+T_{in}(t)}\frac{28.2}{22.4}} \quad (1)$$

$$W_c(t) = K_{fs} * F_s(t) \quad (2)$$

$$W_{pf}(t) = K_{16} * \triangle P_{pa}(t) * M_{pf}(t) \quad (3)$$

$$M_c(t) = W_c(t) - K_{15} * M_c(t) \quad (4)$$

$$M_{pf}(t) = K_{15} * M_c(t) - W_{pf}(t) \quad (5)$$

$$P(t) = K_6 * M_{pf}(t) + K_7(t) * M_c(t) + K_8 \quad (6)$$

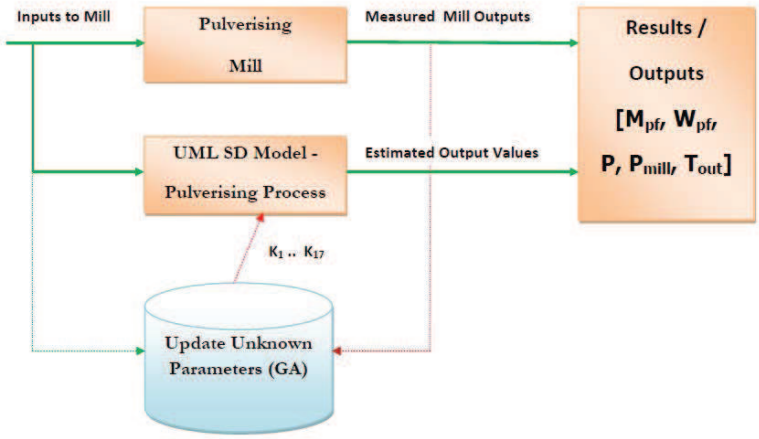

Fig. 4. Online mill performance simulation

$$\triangle P_{mill}(t) = K_9 * \triangle P_{pa}(t) + \triangle P_{mpd}(t) \quad (7)$$

$$\triangle P_{mpd}(t) = K_{11} * M_{pf}(t) + K_{12} * M_c(t) - K_{13} * \triangle P_{mpd}(t) \quad (8)$$

$$T_{out}(t) = [K_1 * T_{in}(t) + K_2] * W_{air}(t) - K_3 * W_c(t) - [K_4 * T_{out}(t) + K_5][W_{air}(t) + W_c(t)] \quad (9)$$

The main input variables supplied to the pulveriser system include raw coal flow into the pulveriser, primary air differential pressure and primary air inlet temperature. The output variables include pulveriser differential pressure, outlet temperature and mil current. The online coal mill optimization process is specified as a constraint file as shown below. During the refactoring process these equations are affixed under the $< body >$ tag elements section of the Pulverising() method call.

The values of the constant co-efficients are obtained using the Genetic Algorithm. The code generation helps the researchers and experienced engineers in analyzing and comparing the simulation results of the online mill model with the real plant data set values and thereby improving the mill performance.

3.2 OCL Constraints in the Coal Pulverising Process

This section presents the approach by specifying the temporal properties associated with the pulverisation process in a coal-fired thermal power plant system. The structural constraints that must necessarily hold true or checked during the pulverization process are represented as invariants using the OCL file.

```
Mc = ((Wc - (k15 * self.Mc)) * T )
Mpf = (((k15 * self.Mc) - (self.Wpf)) * T)
DPmpd = (((k11 * self.Mpf +  (k12 * self.Mc) - (k13 * self.DPmpd))) * T)
```

```
Tout = (((((k1 * self.Tin) + k2) * self.Wair) - (k3 * self.Wc) -
     (((k4 * self.Tout) + k5) * (self.Wair + self.Wc)) +
     k14 * ((k6 * self.Mpf) + (k7 * self.Mc) + k8) +
     ((k17 * self.Tout))) * T) + self.Tout
P = (k6 * self.Mpf) + (k7 * self.Mc) + k8
DPmill = k9 * self.DPpa
Wpf = k16 * self.DPpa + self.Mpf
```

The following are a few constraints associated with the two classes CoalStorage and PowerPlant in the thermal power plant system:

```
context CoalStorage inv cosize:CoalSize=20
context CoalStorage inv hval: HGI=55
```

The invariants that must hold true for the CoalStorage class are:

```
context CoalStorage inv cosize:CoalSize=20
context CoalStorage inv hval: HGI=55
```

The typical invariants that must hold for the Pulveriser class are as shown below.

```
context Pulveriser inv: Wc $<=$ 45
context Pulveriser inv: Wair $<=$ 75
context Pulveriser inv: Tin $<=$ 300
context Pulveriser inv: DPpa $<=$ 180
context Pulveriser inv: outlet temperature $<=$100
context Pulveriser inv: DPmill $<=$ 500
context Pulveriser inv: P $<=$ 60
context Pulveriser inv: if a.Stage = 3 then
  Speed=54 else Speed=52 endif
context Pulveriser inv:if a.Stage = 3
 then self.Type = 'HP803PXBowl'
 else self.Type = 'XRP763BowlRoller'endif
context Pulveriser inv:if CoalHGI = 55 and CoalMoisture = 0.1
  and CoalFineness = 0.7 then
  if a.Stage = 3 then self.Capacity2 = 39.9 else
       self.Capacity2 =33.8
  endif
  else self.Capacity2 $ <>$ 0
endif
```

3.3 Safety and Temporal Constraints in the Coal Pulverizing Process

The temporal constraints involved in the coal milling process include the safety and time related aspects of all the classes or objects participating in that process. For the **CoalStorage** class the main temporal constraints involved are:

1. Failure in the level of the minimum storage level or maximum storage level occurred; that means the coal level exceeds the limit of the maximum storage level or has reached the minimum level.

For the **Pulverizer** class the main temporal constraints to be considered are:

1. The system is in initialization mode until all the physical sub components or units inside the pulveriser are in ready mode and all the external attributes supplied to the system satisfie the normal range values for proper functioning of it or a failure in the level of coal quantity supplied to the pulverizing mill has been identified.
2. The system is in the normal mode, which is the standard operating mode when the program tries to maintain the raw coal level in the pulveriser between the level of values with which all physical units are operating correctly.

The below listed are a few safety and temporal constraints associated with the coal pulverisation process.

When the Pulveriser system is in the initialization mode, it remains in this mode until all physical units are ready or a failure in the pulverising mill has occurred.

```
context Pulveriser inv:
self.mode = # Initialization implies
 always self.mode = # Initialization
 until (Pulveriser Unit.allInstances ->
forAll(u: Pulveriser Unit | u.ready))
```

Instantly the program recognizes a failure in the Coal Mill system until it goes into the rescue mode.

```
context Pulveriser inv:
self.coalmillFailure implies
next self.mode = # Rescue
```

Failure of any of the physical or measured units except the coal mill puts the program in to degrade mode.

```
context Pulveriser inv:
(if Wc > 45 or HGI! = 55 or Wair > 75 or Tin<=300 or DPpa >
 180 or outlet_temp > 100 or DPmill > 500 or P > 60) implies
 next self.mode = # Degraded
```

When the Coal Mill system is in the initialization mode and a failure of the coal mill is detected, it puts the program into emergency stop.

```
 -- Wc<=45
 context Pulveriser inv:
(self.mode = # initialization and self.WcFailure) implies
 next self.mode = # EmergencyStop

 context Pulveriser::openMillValve()
 pre: self.valve.mode = # Off
 post: self.valve.mode = # On

--Coal Storage constraints
context CoalStorage::getCoalLevel(): Double
pre: self.program.mode = # Normal
post: self.coalLevel = result
```

4 Results and Discussions

The UML SD design model for the coal pulverization process is shown in Figure 5:

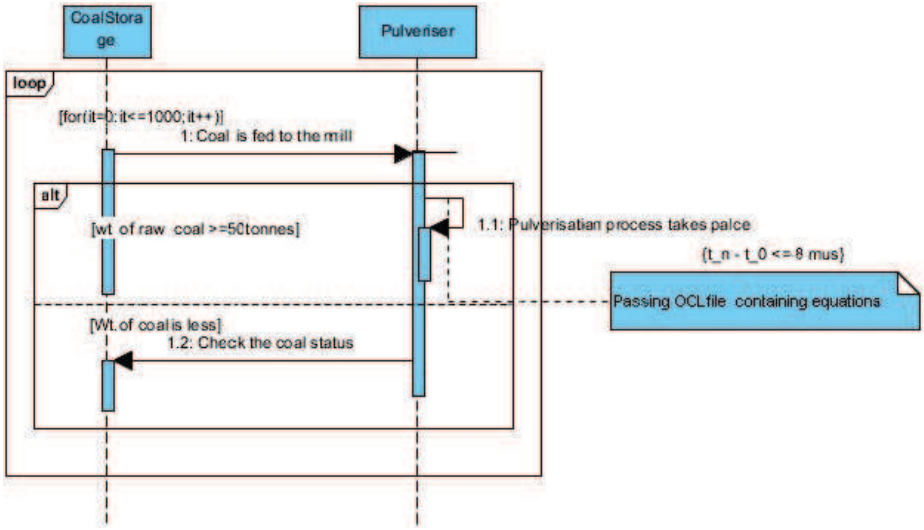

Fig. 5. UML Sequence diagrams

The OCL framework generated for the OCL constraints given below is as shown in Figure 6. Here the CoalStorage class has two constraints associated with it, which are converted into two $< MethodDetails >$ elements under the $< ExtendedConstraints >$ object tag.

The OCL-TOCL framework generated for the mill optimization process is as follows.

The framework generated for the preconditions and postconditions specified as OCLconstraints that must satisfy for the Pulveriser object is:

The code framework generated for invariants of the Pulveriser subsystem is:

5 Conclusion

This paper concentrates on providing a generic framework for refactoring the specification of complex systems modeled using UML2.0 sequence diagrams. It focuses on combining the structural and the behavioral constraints, thereby offering a path for consistent and quality source code generation. The system has been formally modeled using the OCL/TOCL language to provide explicit and precise system information to the design. A generic template framework has been built based on the constraints as well as the UML sequence metadata of

```
<extendedConstraints context="CoalStorage" type="ocl" >
  <MethodDetails name="hval" returntype="boolean" type="inv">
    <body>
      <constraints>
        <constraint>
          <condition exprn="HGI=55"/>
        </constraint>
      </constraints>
    </body>
  </MethodDetails>
  <MethodDetails name="cosize" returntype="boolean" type="inv">
    <body>
      <constraints>
        <constraint>
          <condition exprn="CoalSize=20"/>
        </constraint>
      </constraints>
    </body>
  </MethodDetails>
</extendedConstraints>
```

Fig. 6. The OCL framework generated for the OCL constraints

```
<body>
  <constraints>
    <constraint>
      <condition exprn="Mc = ((Wc - (k15 * self.Mc)) * T )"/>
      <condition exprn="Mpf = (((k15 * self.Mc) - (self.Wpf)) * T)"/>
      <condition exprn="DPmpd = (((k11 * self.Mpf +  (k12 * self.Mc) - (k13 * self.DPmpd))) * T)"/>
      <condition exprn="Tout = (((((k1 * self.Tin) + k2) * self.Wair) - (k3 * self.Wc) - (((k4 * self.Tout) + k5)
      * (self.Wair + self.Wc)) + k14 * ((k6 * self.Mpf) + (k7 * self.Mc) + k8) + ((k17 * self.Tout))) * T) +
        self.Tout"/>
      <condition exprn="P = (k6 * self.Mpf) + (k7 * self.Mc) + k8"/>
      <condition exprn="DPmill = k9 * self.DPpa"/>
      <condition exprn="Wpf = k16 * self.DPpa + self.Mpf"/>
    </constraint>
  </constraints>
</body>
```

Fig. 7. The XCodeFrame generated for the online coal mill optimization process for the Pulveriser subsystem

the system by using refactoring approach. The proposed method facilitates the mathematical verification of pulveriser system in a thermal power plant. The representation of extra information as static as well as temporal constraints attached to certain locations of the objects lifelines in the sequence diagram allows the identification of gaps and contradictory specifications during the source code generation process.

6 Acknowledgments

The authors would like to thank the Control and Instrumentation Group, CDAC, Thiruvanathapuram for providing the raw plant data of pulverising mill which is used for modeling and validating the system. This work is supported by SPEED-IT programme of Kerala State IT-Mission under Govt.of Kerala.

```
<extendedConstraints context="Pulveriser" type="ocl">
<MethodDetails name="Pulverising()" type="function" returntype="Boolean">
<preconditions>
 <constraints>  <constraint>
  <condition exprn="T=1"/>
  <condition exprn="Wc 0 &lt;=WC &lt;=45"/>
   <condition exprn="Wair &lt;= 72"/>
   <condition exprn="Tin  &lt;= 250"/>
   <condition exprn="DPpa &lt;= 180"/>
   <condition exprn="Mc = 0"/>
   <condition exprn="Tout = 0"/>
   </constraint>  </constraints>
 </preconditions>
<postconditions>
 <constraints>  <constraint>
 <condition exprn="Mc =  Mc@pre &lt;=190"/>
 <condition exprn="Mpf =  Mpf@pre &lt;= 190"/>
 <condition exprn="DPmill =  DPmill@pre &lt;= 150"/>
 <condition exprn="Wpf = Wpf@pre &lt;= 45"/>
 </constraint></constraints>
</postconditions>
</MethodDetails>
</extendedConstraints>
```

Fig. 8. The preconditions and postconditions specified as OCLconstraints that must satisfy for the Pulveriser object

```
<MethodDetails name="inv_21" type="inv" returntype="boolean">
 <body>
  <constraints>  <constraint>
   <condition exprn="DPpa&lt;=180"/>
  </constraint>  </constraints>
 </body>
</MethodDetails>
 <MethodDetails name="inv_22" type="inv"
 returntype="boolean">
 <body>
  <constraints>   <constraint>
    <condition exprn="outlet_temp&lt;=100"/>
   </constraint>  </constraints>
  </body>
</MethodDetails>
<MethodDetails name="inv_23" type="inv" returntype="boolean">
 <body>
  <constraints>  <constraint>
   <condition exprn="DPmill&lt;=500"/>
  </constraint>  </constraints>
 </body>
</MethodDetails>
<MethodDetails name="inv_24" type="inv" returntype="boolean">
 <body>
  <constraints>
   <constraint>
    <condition exprn="P&lt;=60"/>
   </constraint>
  </constraints>
 </body>
</MethodDetails>
```

Fig. 9. The code framework generated for invariants of the Pulveriser subsystem

References

1. Unified Modelling Language, http://www.uml.org/.
2. Object Management OMG. Unified Modeling Language Specification 2.0: Infrastructure. Technical Report ptc/03-09-15, OMG (2003).
3. G. Booch, J. Rambaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley (1999).
4. Model Driven Architecture, White Paper, Object Management Group OMG, May (2014).
5. XML Metadata Interchange, www.omg.org/spec/XMI/
6. OMG, Object Constraint Language (OCL) Specification, version 2.0,2006 http://www.omg.org/spec/OCL/2.0/
7. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, PhD thesis. Univ. of Illinois (1992).
8. Roberts, D.: Practical Analysis for Refactoring. PhD thesis. Univ. of Illinois (1999).
9. Fowler, Martin: Refactoring: Improving The Design of Existing Code. Pearson Education India (1999).
10. Astels, Dave: Refactoring with UML. In: 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, 67–70 (2002).
11. Suny, G., Pollet, D., Le Traon, Y., Jzquel, J. M.: Refactoring UML models. In International Conference on Unified Modeling Language UML -2001. LNCS 2185. Springer Berlin Heidelberg, 134–148 (2001).
12. Maddeh, M., Romdhani, M., Ghdira, K.: Classification of Model Refactoring Approaches. Journal of Object Technology, 8.6, 143–13958 (2009)
13. Mens, Tom, Tourw, Tom: A survey of software refactoring. Software Engineering, IEEE Transactions on 30.2, 126–139 (2004).
14. Mens, Tom, Taentzer, Gabriele, Mller, D.: Model-driven Software Refactoring. Model-Driven Software Development: Integrating Quality Assurance, 170–203 (2008).
15. Mens, Tom, Taentzer,Gabriele, Mller, Dirk.: Challenges in model refactoring. In: 1st Workshop on Refactoring Tools, University of Berlin. Vol. 98 (2007).
16. Chitra M. T., Elizabeth Sherly.: Refactoring sequence diagrams for code generation in UML models. In IEEE Int. Conf. on Advances in Computing, Communications and Informatics ICACCI, 2014, 208–212 (2014).
17. Folli, Alessandro, Mens, Tom.: Refactoring of UML models using AGG. Electronic Communications of the EASST (2008).
18. France, R., Chosh, S., Song, E., Kim, D. K.: A metamodeling approach to pattern-based model refactoring. Software, IEEE, 20(5), 52–58 (2003).
19. Y. G. Zhang, Q. H. Wu, J. Wang, G. Oluwande D. Matts, and X.Zhou.: Coal Mill Modeling by Machine Learning Based on Onsite Measurements, Energy Convers. IEEE Trans., 17(4), 549–555 (2002).
20. J. Wei, J. Wang and Q. H. Wu.: Development of a Multisegment Coal Mill Model Using an Evolutionary Computation Technique, IEEE Trans.Energy Convers., 22(3), 718–727 (2007).