Using a Formal Language Constructs for Software Model Evolution

Samuel A. Ajila, PhD., P.Eng
Department of Systems and Computer Engineering,
Carleton University
1125 Colonel By Drive, Ottawa, ON K1S 5B6
ajila@sce.carleton.ca

Shahid Alam
Department of Computer Science,
University of Victoria
3800 Finnerty Road Victoria BC V8W 2Y2
salam@cs.uvic.ca

Abstract—In this paper we present an automated support for software model evolution using a formal language constructs. For this, we extended Object Constraint Language (OCL) with actions to define a new language - CAL (Constraint with Action Language), which gives users the ability to evaluate objects change a-priori. We have added a data type, directed acyclic graph (DAG) to CAL to automate model evolution. DAG has been annotated (ADAG) with dependency weights based on an improved graph labeling scheme. Rules are designed to translate a UML model into CAL data structure (ADAG). CAL contains a small set of constructs, but is powerful enough to be used efficiently for typical software evolution management operations like impact analysis, correction, improvement and enhancement of models. A prototype tool VCAL (visual CAL), for dependency analysis of UML Class Diagrams is presented.

Keywords—software model evolution; OCL; DAG; UML; objects change; impact analysis

I. INTRODUCTION

Complex software system evolves over time and its architecture will change as the system meets the challenges of its environment. Empirical research shows that it is very difficult to completely predict software changes and its costs [3], [4], [10]. More time and money are spent maintaining a software system than on its original development. Also, software evolution is a change process that covers the life cycle of a software product. According to the European Research Consortium for Informatics and Mathematics (ERCIM) working group on software evolution [12] “The only way to overcome or avoid the negative effects of software aging is by placing change in the center of the software development process.” Therefore, there is need for a better and automated support for software evolution at the modeling level. Such support must be attacked from different perspectives [1], [12]: (i) basic research on techniques to understand, manage, and control software change; (ii) development of models, languages, tools, and methods to provide support for software model change; and (iii) validation of complex industrial real world software case studies [1], [2].

The aim of this research therefore is to develop an automated support for a bi-directional evolution (or co-evolution) between different representations of software artifacts (models). That is, “... placing the change in the centre of the software development process. [2][12]” Specifically, we want to research the automation of model evolution using a formal modeling language constructs and graph theory to support co-evolution.

Two of the top challenges identified by ECRIM [12] were to raise the level of abstraction (i.e. model evolution) and automated support for model evolution through modeling languages and tools. Support for software evolution is advancing in many ways but it is observed that almost all existing tool supports for software evolution is primarily targeted to source code and at best are software life cycle domain specific. In particular, design and modeling phases supported by UML and, UML CASE tools typically provide less support for software evolution. Since the advent of model-driven software engineering, the urgent need for more relevant techniques and tools for model evolution has increased. We are now seeing a move towards serious research attempts in model evolution. For instance, model driven architecture (MDA) allows for the separation of a platform independent model (PIM) from a platform specific model (PSM). Executable UML takes the concept of MDA further by removing the PSM from MDA and directly generates code [7]. To support executable UML, we need to fully introduce evolution at the model level. UML has notations, symbols and stereotypes to define dependency relationships, but there is no modeling technique for automated software evolution management [14]. Neither Object Constraint Language (OCL) [15] nor any other specification language provides direct support for model evolution. Recent advances include model re-factoring [7], impact analysis and change management of UML models [3], and model evolution with aspect-oriented mechanisms [16]. Earlier theoretical and foundational research works include software evolution theory and feedback by Leman et al. [9], and Madhavji et al. [10], and the work on the extension of meta-models in UML 1.3 based on stereotyping [14] and the application of action semantics to UML 1.4 [13].

The long term goal of this research is to develop techniques that will support automatic software model evolution. Specifically, our objectives are to:

Develop Support for model evolution and co-evolution: A challenge here is the necessity to achieve an automatic co-evolution among different representations of software models. Modification in one type of model should always be reflected by corresponding changes in other related models to ensure consistency of the software.

Using language constructs to automate model evolution: A modeling language should provide a more direct and
explicit support for software evolution. This will allow object change to be treated as a first-class entity in the language. Firstly, changes can be introduced a-priori into the system without impacting the current representation of the system under development. In this case, changes are only visible after the software modeler is satisfied with the computation of the impact of such changes. Secondly, the modeling language construct can be used to control bi-directional evolution between two different representations of the software system. Thirdly, the modeling language construct can be used as a means of verification and validation using formal methods. This way, it is not imperative to verify the entire software design model (and source code) because of a small change to the specification of the software.

The roadmap for achieving this goal consists on using a formal modeling language constructs to achieve some level of model evolution. The idea is to extend Object Constraint Language (OCL) by adding actions to it. OCL is normally used as a query language for UML models. Expressions and rules in OCL can be used to identify changes in a model but cannot be used to change the state of the system. Therefore, actions (now part of UML 2.0 semantics) are needed in addition to constraints in OCL to be able to apply changes to the design model. Furthermore, we need tools for model transformation and model visualization. In this case, we have a choice between direct model manipulation or transformation language. The first choice is tedious to implement and not very good for co-evolution of models. A transformation language provides a means to explicitly represent transformation behavior and more concise description of model elements than is possible in direct manipulation. Like the model evolution language, the model transformation language must have the power to represent all types of modeling objects and behaviors including querying, modification, and navigation. For this, we use a directed acyclic graph (DAG) as a transformation language. In this case, models are treated as graphs and sub-models as sub-graphs (with the same properties as a graph). Transformation rules are developed and apply as rules for rewriting graphs. Each element in the model is a node in the graph and the properties attached to the model elements are attached to the corresponding node in the graph. Rules are executed using pattern matching techniques based on set of dependency relationships among models and model elements.

The rest of this paper is organized as follows. Section 2 gives the background and related works. In section 3, we present model transformation using ADAG and in section 4, we define CAL. Section 5 presents a prototype – VCAL and we give our conclusion in section 6.

II. BACKGROUND AND RELATED WORK

A. Background

The OCL standard library consists of built-in types, predefined types and operations [15]. OCL collection is a predefined type. There are four collection types in OCL - Set, OrderedSet, Sequences and Bag. A Set is a mathematical set. An OrderedSet is a Set whose elements are ordered. A Bag is a Set that may contain duplicate unordered elements. A Sequence is a Bag but the elements are ordered. These collections have an extensive set of operations for conversion, comparison, query, access, selection and iteration. In order to formally specify the internal model of the actions and operations in CAL for the tool that implements them and to verify and reason about the function and behavior of these actions and operations – we have used a specification language called TLA [8], the Temporal Logic of Actions.

TLA+ is a tool for specifying the mathematical model of a system, which may have some behavioral properties. These properties can be functional and/or logical, and represent a correct execution of the system. Since TLA+ specification is a formal description of the system, tools can be applied to find errors and test the system. TLC, a TLA+ model checker [8], is a tool that is written for this purpose and is used in this paper to verify the specifications of CAL actions and operations. The CAL actions and operations can be directly specified in TLA+ but may need significant amount of space and time. To speed up this process and save space, we have used an algorithmic language that is based on TLA+ named plascal. TLC translates specifications in plascal to TLA+.

B. Related Work

One of the early efforts to automate software evolution in UML was in 2000 when Tom Mens and Theo D’Hondt [11] proposed to extend the metamodel of UML 1.3 and based on this extension defined evolution contracts using stereotypes such as <<add>>, <<remove>>, <<connect>> and <<disconnect>> etc. The purpose of these contracts was to automatically detect conflicts.

Briand et al. [3] proposed an approach to perform impact analysis of UML model elements and present promising empirical results. This is the first empirical effort at automating impact analysis of UML models, but it does not address making changes to the model. The impacted elements, whether direct or indirect, are determined using impact analysis rules defined in OCL. The use of constraints, as rules, defined in OCL by itself is not optimal for impact analysis. Invoking constraints each time for tracing the change of a UML model element is not efficient. For example, select and equals have been used to query the model and to match the pattern respectively. The approach described needs to compare two complete models to detect changes. That is, the original model and the changed model are required to complete the detection process. Our approach works on just one model. As the model is changing the tool is analyzing the change, which makes the analysis process dynamic and compact.

III. MODEL TRANSFORMATION USING ADAG

We need a transformation language for model evolution and this language must have power to represent all types of modeling objects and behaviors such as querying, modification and navigation. For this, we have used a Directed Acyclic Graph (DAG). Therefore, ADAG is Annotated DAG. In this section we present ADAG, ADAG
labeling scheme, and rules for translating UML model to ADAG.

A. ADAG labeling

Effective graph labeling scheme is very important in the use of DAG as a model transformation language. In this paper, we introduced a labeling scheme called dependency relationship weight (DRW). DRW is based on DAG-Lite [17] with improvement in label size and application to graph. In general, the following characteristics are required in a DAG labeling scheme:

- Size of the label. It should be as compact as possible;
- Time it takes for the queries to complete the DAG operations, for dependency analysis;
- Updates should require minimal re-labeling;
- Reachability: checks if one of the vertex of a DAG is reachable from (dependent on) other vertices, i.e. to find out if a path exists between two vertices;
- Descendants: find all the descendants of a vertex in a DAG, i.e. which of all other vertices depend on the vertex.

We formally define DAG, G, as follows, where V is the set of vertices and E the set of edges:

\[ G = (V, E, L, W) \]

\[ E = \{ (x, y) \mid x \in V, y \in V, xy \in E \} \] such that E is a Partial order on set V

\[ L = \{ l(x) \cup l(y) \mid x, y \in E \} \]

\[ W = \{ w(x, y) \in E \mid w(x, y) \text{ is the weight of } x \} \]

Total weight of \( v \in V \) is defined as

\[ w(v) = \sum w(a, b), \text{ where } D(v) \in E \text{ and is a set of descendants of } v \] and \( (a, b) \in D(v) \)

We also define Indegree of a vertex as the number of edges adjacent to that vertex and outdegree is defined as the number of edges adjacent from a vertex. Ideally, for any two vertices we should be able to determine reachability by comparing only their labels. But, for a graph of more than 100 vertices it is computational expensive to calculate and extremely difficult to visually look for reachability between two distant vertices. As an improvement, our DRW used uniqueness of prime factorization – a property of prime for a vertex.

Let \( a, b \in \mathbb{N} \) and \( \text{LCM}[a, b] \) as the least common multiple of \( a \) and \( b \), the set of children of \( v \in V \), \( C(v) = \{ v_1, v_2, v_3, ..., v_n \} \), where \( 1 \leq n \leq \text{outdegree}(v) \) and \( np \) is the next prime from an ordered list of primes starting with 2. Then we define two equations for labeling the DAG as follows:

\[ L(v) = \begin{cases} \text{outdegree}(v) = 0 & \text{if } n \\ \text{LCM}[\text{label}(v_1), \text{label}(v_2), ..., \text{label}(v_n)] & \text{else} \end{cases} \]

\[ L(v) = \begin{cases} L(v) \times np & \text{if } \text{outdegree}(v) = 0 \\ \text{LCM}[\text{label}(v)] & \text{else} \end{cases} \]

The complete label of DAG in Figure 1 using equations (1) and (2) is shown in Figure 2. All the leaves, A, B, C, D, E and M are labeled as 1. The prime numbers used are underlined. We can find the reachability between vertex Q and vertex I by simply computing the mod of the label of vertex Q with the label of vertex I as \( L(Q) \mod L(I) = 1385670 \mod 7 = 1 \). Therefore, vertex I is not dependent on vertex Q. We can find all the descendants of a vertex by computing the mod of each vertex's label with the label of the vertex. If the mod is 0 then a descendant of the vertex is found and vice versa. For vertex P in Figure 2 we find the following set of descendants: \( D(P) = \{ O, L, K, F, G, H, A, B, C, D, E \} \). So, using prime numbers from an ordered list of primes starting from 2, we can calculate the maximum number \( M \) that can be used for a label in the DAG, using the following equation:

\[ M = \prod p_i \]

\[ i = 2 \]

Where \( m \) is the maximum prime number used and \( p \) is the prime factor of \( m \). The size of \( M \) depends on \( m \). For a DAG with \( n \) vertices, \( m \) can be calculated for worst and average cases, using equation (1) as follows:

Worst case, \( m = n - 1 \): In this case, a DAG is a straight line, the outdegree(\( v \)) > 0, and label of each vertex is equal to its child's label, so each vertex is multiplied by \( np \).

Average case, \( m = n / 2 \).

In DRW, not every vertex in the DAG is multiplied by an exclusive prime number. In equation (1), \( L(v) \) is only multiplied by \( np \), when it equals one of the labels of children of \( v \) and hence reduces the size of \( M \) in equation (3). This significantly reduces the number of digits in the label as compared to a similar scheme used in DAG–Lite [17], which assigns an exclusive prime number that is multiplied by the label of all the vertex's parents, as defined in equation (1). We have also improved the space for storing all the labels of the node in a DAG by assigning 1 as the label of leave vertices as compared to the DAG-Lite...
scheme, which assigns an exclusive prime number to a leave, which is multiplied by the label of all the leave's parents.

Figure 2. Labeled Sample DAG

B. Translating a UML model to ADAG

We used the definition of DAG given in section 3 to define rules for translating a UML model to ADAG and an example of UML class diagram translation rules is given below:

If \( V \) is the set of vertices and \( E \) the set of edges in the graph, \( m, x, y \) and \( z \) are model elements, and \( R_D \in \) (Dependency \( \lor \) Composition \( \lor \) Generalization \( \lor \) Realization), then

\[
\begin{align*}
R1.1: & \ V_m \ \{ \ (m \in V) \land \neg (\neg m \in V) \} \\
R1.2: & \ V_{xy} \ \{ \ (xy \in E) \rightarrow ((x R_D y) \lor (y R_D x)) \} \\
R1.3: & \ V_{xy} \ \{ \ (x R_D y) \rightarrow (y R_D x) \} \\
R1.4: & \ V_{xy} \ \{ \ ((x R_D y) \land (y R_D z)) \rightarrow (x R_D z) \} \\
R1.5: & \ V_{xy} \ \{ \ ((x R_D y) \land (y R_D z)) \rightarrow ((x R_D z) \rightarrow (w(x,y) = 1)) \lor (R_D \in \) (Generalization \( \lor \) Composition)) \rightarrow (w(x,y) = 2)) \} \\
\end{align*}
\]

According to rule R1.1 model elements become vertices of the graph. Rule R1.2 states that if there is a relationship between two model elements then they become an edge in the graph. The relationship \( R_D \) is described by Rules R1.3 and R1.4 as transitive and non-reflexive respectively, so by definition the graph is a DAG. Rule R1.5 assigns a self-weight to each vertex in the edge. This assignment of self-weight to the model elements is due to the couplings of different relationships among the model elements.

IV. CONSTRAINT AND ACTION LANGUAGE - CAL

CAL is a text based scripting language defined for the purpose of model evolution. For this, OCL is extended by adding 5 actions, 10 operations and 1 data type. Together with ADAG, which is also the CAL data structure, the actions and operations are used to automate dependency analysis of a UML model and can be used for impact analysis. The five action statements are createStmt, deleteStmt, addAttrStmt, createLinkStmt and deleteLinkStmt. As the names suggest, they can be used to create or delete objects or attributes and create or delete links or associations between elements. Given below are CAL production rules.

Using OCL syntax, we describe in Table 1 the informal semantic of the CAL library and CAL actions in Table 2. The word result is used to refer to the value that is obtained from evaluating the operation and the word iterate is any searching technique (e.g. depth first or breadth first, etc.) suitable for a directed acyclic graph. Post conditions are used to describe the properties of the result.

Like OCL, CAL is a typed language, so we have defined a collection type, DagType, for the CAL data structure. In addition, CAL contains a library function, Dag (cf. Table 1). This function is used to create ADAG of a UML model (cf. Table 2). The operations depend and descendants check for dependency between two elements and all the descendants of an element respectively. Also, depend is used in the production rules deleteStmt and deleteLinkStmt to check the dependency, and accordingly the dependent objects are deleted or unlinked.

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DagType (data type)</td>
<td>inv: self.name = 'Dag(' + self.element.name + ')’</td>
</tr>
<tr>
<td>Dag (operation)</td>
<td>Dag is a DAG of an instance of the context. The instance becomes the root vertex, and all its dependent vertices become the child vertices in the form of a DAG.</td>
</tr>
<tr>
<td>isdag (operation)</td>
<td>Context Dag :: isdag() : Boolean post: result = self-&gt;type-&gt;elementType-&gt;oclIsKindOf(DagType) = 0</td>
</tr>
</tbody>
</table>
The complete process of using the ADAG can also be displayed visually using one of the CAL library operations, view, to let the software design or maintenance engineer decide where and what changes are to be made to the model.

CAL actions, create, delete, link, unlink and addattr, are responsible for incrementing the global variable CHANGE_RATE (defined in the CAL library). CHANGE_RATE is the number of objects changed and can be used to calculate the percentage of objects that change from one version to the next [10]. Similarly create and delete increment or decrement the global variables OBJECTS_ADDED and OBJECTS_REMOVED respectively to calculate the growth rate [10], which is the difference between added and removed objects.

The complete process of using the ADAG can also be displayed visually using one of the CAL library operations, view, to let the software design or maintenance engineer decide where and what changes are to be made to the model.

CAL actions, create, delete, link, unlink and addattr, are responsible for incrementing the global variable CHANGE_RATE (defined in the CAL library). CHANGE_RATE is the number of objects changed and can be used to calculate the percentage of objects that change from one version to the next [10]. Similarly create and delete increment or decrement the global variables OBJECTS_ADDED and OBJECTS_REMOVED respectively to calculate the growth rate [10], which is the difference between added and removed objects.

1 = Checks the presence of an element in the ADAG
2 = Computes the mod of two labels (integers)
3 = Displays an interactive ADAG

TABLE II. CAL ACTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Context Dag :: create (object : T) : Boolean</td>
</tr>
<tr>
<td></td>
<td>pre: self-&gt;isdag()</td>
</tr>
<tr>
<td></td>
<td>body: self-&gt;add(T)</td>
</tr>
<tr>
<td></td>
<td>post: result = self-&gt;includes(T)</td>
</tr>
<tr>
<td>delete</td>
<td>Context Dag :: delete (object : T) : Boolean</td>
</tr>
<tr>
<td></td>
<td>pre: self-&gt;isdag() and self-&gt;includes(T)</td>
</tr>
<tr>
<td></td>
<td>body: if (self-&gt;descendants(T)-&gt;size() = 0)</td>
</tr>
<tr>
<td></td>
<td>then self-&gt;setweight(null, T) and self-&gt;relabel(T)</td>
</tr>
<tr>
<td></td>
<td>else self</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
<tr>
<td></td>
<td>post: result = self-&gt;excludes(T)</td>
</tr>
<tr>
<td>link</td>
<td>Context Dag::link(object : A, object : P, object : C) : Boolean</td>
</tr>
<tr>
<td></td>
<td>pre: self-&gt;isdag() and self-&gt;includes(C)</td>
</tr>
<tr>
<td></td>
<td>body: self-&gt;setParent(P, C) and self-&gt;setweight(A, P) and self-&gt;relabel(P)</td>
</tr>
<tr>
<td></td>
<td>post: result = self-&gt;depend(P, C)</td>
</tr>
</tbody>
</table>

1 = Deletes an element from the ADAG
2 = Labels all the ancestors of an element
3 = Sets the parent of an element (in this case P becomes the parent of C)
4 = Sets the self-weight of an element in the ADAG
5 = Removes the direct link between two elements in the ADAG
6 = Sets the attribute of an element (in this case A is added as an attribute to C)
In this paper, we have presented an approach using a formal language constructs to support software model evolution. Our approach provides a more direct and explicit support for software evolution and allows object change to be treated as a first-class entity in the modeling language. The constraint and action language (CAL) can be used to manage bi-directional software evolution. Our transformation language (ADAG) provides a means to explicitly represent transformation behavior with concise description of model elements that is possible with direct manipulation. Not shown in this paper (for lack of space) are the specifications of eight CAL operations: isempty, isdag, includes, excludes, depend, descendants, flatten and view and six CAL sub-operations: relabel, setweight, CheckDependency, Label, LCM and CheckCycle. For example, the complexity of the CAL operations descendants for worst case and average case is $O(n)$ and depend, have worst case and an average case of $O(n)$ and $O(1)$ respectively. All these operations and actions are verified by the TLC model checker. We have also developed a prototype VCAL that generates dependency and performs impact analysis with little input from users. We are currently developing mapping rules and specifications for UML state machines. We are also exploring the possibility to auto-generate and verify CAL interpreter from CAL specifications using compiler-compiler principle.

ACKNOWLEDGMENT

This paper is a part of research work supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) discovery research grant number RGPIN/250271

REFERENCES