Supporting Automatic Model Inconsistency Fixing*

Yingfei Xiong[1], Zhenjiang Hu[2], Haiyan Zhao[3], Hui Song[3], Masato Takeichi[1], and Hong Mei[3]

[1]Department of Mathematical Informatics
University of Tokyo
Tokyo 113-8656, Japan
xiong@ipl.t.u-tokyo.ac.jp
takeichi@mist.i.u-tokyo.ac.jp

[2]GRACE Center
National Institute of Informatics
Tokyo 101-8430, Japan
hu@nii.ac.jp

[3]Key Lab. of High Confidence Software Technologies
(Peking University)
Ministry of Education
Beijing 100871, China
zhhy@sei.pku.edu.cn
songhui06@sei.pku.edu.cn
meih@pku.edu.cn

ABSTRACT

Modern development environments often involve models with complex consistency relations. Some of the relations can be automatically established through "fixing procedures". When users update some parts of the model and cause inconsistency, a fixing procedure dynamically propagates the update to other parts to fix the inconsistency. Existing fixing procedures are manually implemented, which requires a lot of efforts and the correctness of a fixing procedure is not guaranteed.

In this paper we propose a new language, Beanbag, to support the development of fixing procedures. A Beanbag program defines and checks a consistency relation similarly to OCL, but the program can also be executed in a fixing mode, taking user updates on the model and producing new updates to make the model satisfy the consistency relation. In this way Beanbag significantly eases the development of fixing procedures. In addition, a Beanbag program is also guaranteed to be correct with respect to the three correctness properties we define. We evaluate Beanbag over a set of MOF and UML consistency relations and the result shows that Beanbag is useful in practice.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications - Specialized application languages; D.2.2 [**Software Engineering**]: Design Tools and Techniques

*This work is partially sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703, National Science Foundation of China under Grant No.60873059, 60821003, Japan Society for the Promotion of Science (JSPS), Grant-in-aid for Scientific Research (A) 19200002, and a Grand Challenge project at National Institute of Informatics.

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.

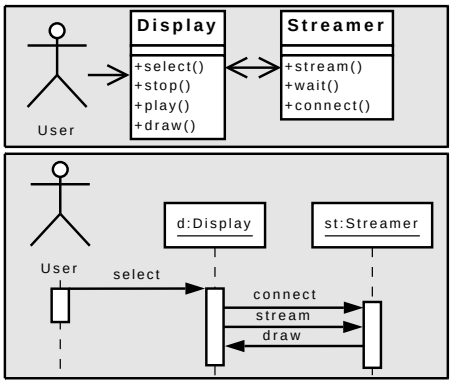

Figure 1: Model of a Video-On-Demand System

General Terms

Design, Languages

1. INTRODUCTION

Modern software development environments often involve models with complex consistency relations. Figure 1 shows an example adapted from [4]. The UML model in Figure 1 describes a client component of a video-on-demand (VOD) system and consists of a class diagram and a sequence diagram. The class diagram describes the static structure of the system and the sequence diagram shows how objects interact with each other through message.

Many consistency relations need to be established over the two diagrams to ensure the correctness of the model. As an example, we give two consistency relations written in the Object Constraint Language (OCL) [12] as follows.

```
C1: context Message
    inv let rec = self.receiver in
        let ops = rec.base.operations in
        ops->exists(oper | oper.name = self.name)
C2: context Message
    inv self.sender <> null and self.receiver <> null
```

C1 requires every message in a sequence diagram to be declared as an operation in the receiver's class. The `context` keyword states that it will be applied to every `Message` object. The concrete relation definition starts from a message (`self`), finds the receiver object (`rec`), finds the operation set in the class of the receiver object (`ops`) and check if there

exists an operation having the same name of the message. C2 requires every message to have a sender object and a receiver object by disallowing related features to be `null`.

These consistency relations would be violated if some parts of the model are modified, and we need tool support to fix the inconsistency [11]. Although it has been recognized that not all consistency relations need to be always established [2] and not all inconsistencies are suitable to be automatically fixed [3], many useful consistency rules can be them always established by propagating the update to other parts when users update part of the model. For example, IBM Rational Software Architect (RSA), a commercial UML modeling tool, deletes all connected messages when a class instance in a sequence diagram is deleted, and thus keeps C2 always established.

Existing automatic fixing approaches [8] require tool developers to manually implement "fixing procedures" specifying actions to take when a change happens. Such a procedure takes user updates as input, and takes actions to propagate the updates to other parts to fix inconsistencies on the model. However, as usually many types of changes may violate a consistency relation, fixing procedures may be very large and thus are very difficult to implement in practice. For example, although IBM RSA implements a fixing procedure for C1, it fails to provide an action for changing the receiver of a message. When a user makes such a change, the tool will report an error. Furthermore, it is difficult to verify that the actions taken by a fixing procedure lead the data to a consistent state.

As a matter of fact, if we examine the definitions of consistency relations, we would find that some relations have already implicitly included the fixing procedure. Consider a simple relation that two primitive values `a` and `b` are equal: `a=b`. If a user changes `a`, the only reasonable action to take is to change `b` accordingly. The same strategy also works when `b` is changed. If we can automatically derive fixing procedures from the consistency relation, we can avoid the problems of manual implementation. However, it is not easy to automatically derive fixing procedures beyond the simplest relations. First, many consistency relations, like C1, have multiple, even infinite numbers of actions to take for some updates. If users change a message name to a new name and cause an inconsistency, a modeling tool can take several actions to fix the inconsistency: creating a new operation in the class diagram, renaming an existing operation in the class diagram, or even changing the receiver of the message to a new class instance containing the new operation. How to choose one among them is unknown. Second, consistency relations may be composed by operators like `and`, `or` and quantifiers. It is unclear how to compose the fixing procedures accordingly while ensuring a correct fixing behavior.

In this paper we suggest a compromising approach. Instead of deriving fixing procedures purely from consistency relations, we add extra semantics to the relation language, so that when a user describes a consistency relation, he also uniquely describes a fixing procedure. Concretely, our contributions can be summarized as follows.

- We design a language, Beanbag, for users to write a consistency relation and a fixing behavior at the same time. The Beanbag language defines consistency relations in OCL-like syntax, but every relation in Beanbag also has a fixing semantics describing when some parts of the data are changed by users, how to change the other parts to ensure consistency. For relations with multiple fixing behavior, Beanbag provides users with more than one way to construct one relation, where a different way indicates a different fixing behavior.
- We propose three properties to characterize the correctness of a fixing procedure. The three properties are STABILITY, PRESERVATION and CONSISTENCY, and are mainly based on the study of bidirectional transformation properties [19] and our previous work on model synchronization [23]. Any Beanbag program satisfies the three properties.
- We have implemented Beanbag and evaluated its expressiveness and usability by developing Beanbag programs for consistency relations in the models of MetaObject Facility (MOF) [15] and UML [3, 4] models. The evaluation shows that Beanbag greatly eases the development and can support many useful scenarios in practice. In addition, a preliminary version of Beanbag [22] has already been used by several research groups developing applications like domain-based UML synchronizer [16] and software product line.

To have a concrete idea of Beanbag, let us write a Beanbag program for relation C1. A Beanbag relation describing C1 can be defined as follows.

```
def C1(msg, model) =
 let rec = model.(msg."receiver") in
 let opRefs = model.(rec."base")."operations" in
 opRefs->exists(opRef | model.opRef."name"=msg."name")
```

We can see that the Beanbag program is very similar to the OCL expression. One small difference is that we need to write `model.(msg."receiver")` instead of `msg."receiver"`. This is because of the data structures used in Beanbag, and is not important for now.

To apply this relation to all instances of `Message`, we further write the following code:

```
def C1onAll(model, meta) =
 model->forall(obj |
  isTypeOf(obj, "Message", meta) and C1(obj, model)
  or not isTypeOf(obj, "Message", meta)
)
```

The `C1onAll` relation works in two modes. In the checking mode, it runs as a normal OCL expression, taking the current model and the meta model as input and producing a boolean value to indicate whether the relation is satisfied. In the fixing mode, it takes as input the current model, the meta model and the updates that users try to apply to the model, and produces new updates representing actions to take to make the model consistent. The input updates can be a single update, changing a single feature or inserting/deleting an object. It can also be a combination of several updates performed by different users in a distributed environment, changing several locations or inserting/deleting several objects. We will give the concrete syntax for describing updates in Section 3.

Putting it more concretely, the fixing mode of `C1onAll` proceeds in a similar way to the checking mode, but propagates updates when it encounters one. Suppose a user has renamed an operation to a new name. `C1onAll` will invoke `C1` on all `Message` objects and `C1` will check if there exists an operation with the same name. For the renamed message, such an operation cannot be found. Then the `exists` statement will insert a new `null` reference in the collection and proceed to the inner relation. The expression `model.opRef` will create a new operation and replace the `null` reference

with the actual reference. Finally, the equality relation will assign the changed name to the newly created operation. We will see the precise semantics in Section 5.3.

Now if we want to rename an existing operation instead of inserting a new one in the fixing mode, what we need to do is to change `exists` to `exists!` in the last line. The new Beanbag program runs the same in the checking mode, but in the fixing mode it will rename the operation that originally corresponded to the message.

Note that the current program for C1 can be improved; the current version will insert a new operation even when we change the receiver of a message or when we delete/rename an operation in the class diagram. We can describe a more natural fixing behavior by extending C1 to allow the message name to be `null`. We will see a fully featured program in Section 5.4.

The rest of the paper is organized as follows. First, Section 2 discusses related work. Section 3 describes how we represent data (models) and updates to build up the foundation for formal discussion. Based on that, Section 4 defines the correctness properties of a fixing procedure. After that, Section 5 presents the Beanbag language, including the syntax, the semantics of the checking mode and the fixing mode, and a few examples to show how to write Beanbag programs in practice. Section 6 discusses the correctness and termination of Beanbag. Section 7 evaluates the expressiveness and usability of Beanbag through practical cases. Finally, Section 8 concludes the paper.

2. RELATED WORK

Many approaches provide support for automatic inconsistency resolution from logic perspective. These approaches range from more theoretical, first-order logic [6] and description logic [20], to more practical, OCL expressions [18]. In general, these approaches require developers to write a set of fixing rules in the "condition↦action" form. The system checks the condition of each rule, and executes the action when the condition is satisfied. It takes a lot of efforts to define a full set of fixing rules, and there is no guarantee that the actions will bring the data to a consistent state. Compared to them, our approach exploits the information in the user updates and in the consistency rules. The Beanbag language is much easier to write and the fixing behavior of every Beanbag program is guaranteed to be correct.

Grundy et al. [8] propose a general framework for managing inconsistency in multiple-view software. Similar to us, their framework uses updates as a start point to fix inconsistency. However, they still require developers to write fixing actions for each type of updates. As a result, the problem of development cost and the problem of correctness still exist. More recent work [10] from the same group uses spreadsheet-like mechanism to define the relation over model and thus the fixing process is automated as reevaluating the cell expressions. However, this approach only propagates updates in one direction and cannot help if the location updated by users is the result of some expression.

Some approaches seek for automated means to generate a set of fixing actions from logical expressions. Typical approach includes the white-box analysis of first-order logic [11] and the black-box analysis of the consistency rules [4]. Compared to ours, these approaches generate the fixing actions purely from a consistency relation, but require human interventions in executing the actions, by specifying some locations to fix, choosing one among a set of actions or filling some missed parameters. We believe both types of approaches are important to consistency management, because while some consistency relations are suitable to be established all the time through automatic fixing, some consistency relations are suitable to be manually resolved by humans.

value	::= *primitive* | *dictionary*
primitive	::= `NUMBER` | `STRING` | `BOOLEAN` | `null`
dictionary	::= {*entries*} | {}
entries	::= *entry* | *entry*, *entries*
entry	::= *primitive* –> *value*

Figure 2: Syntax of data

Another branch of related work is bidirectional model transformation approaches, such as QVT [14] and TGG [17]. These approaches establish the consistency relation between two models by using one program to describe two transformations, each update a model according to the other model [19]. As a result, bidirectional transformation cannot deal well with inconsistency in one model. Our approach is built on the same idea of using one language to describe both the consistency relation and the fixing behavior, but we exploit user updates to fix inconsistency in one model. A sub branch of bidirectional transformation work is model synchronization [1, 23, 9], which also exploits user updates for synchronization. However, existing model synchronization approaches still focus on synchronization of two models, and do not fix inconsistency in one model.

One traditional work on consistency management is the constraint satisfaction problem (CSP) [21]. Approaches to CSP try to find a set of values that satisfies a given set of constraints (consistency relation). Since the problem requires searching the whole state space, it is very difficult for a solution to scale up. Compared to CSP approaches, Beanbag is a more lightweight approach in the sense that we require users to describe the fixing behavior in the Beanbag program, and thus does not suffer from the scalability problem.

Repairing broken data structures [5] is also loosely related. This work dynamically repairs faults in data at runtime according to the consistency relations implicitly specified by the assertions in code. The core to this work is a set of heuristics to fix inconsistency. Different from this work, our approach aims at providing a clear, predictable fixing semantics, so that end users can clearly know how their updates affect other parts of the model.

3. DATA AND UPDATES

In this section we build up the formal foundation by defining data (models) and updates in Beanbag.

Data Although our target is to deal with models, the core Beanbag is built upon a small set of dictionary-based data types. Dictionaries can be used to represent many different kinds of data structures as studied by Foster et al. [7]. In this way we can make our language compact while retaining its expressiveness.

Figure 2 shows the syntax of Beanbag data. There are two types of data (values). One is *primitive* values including numbers, strings, booleans and a `null` value, and the other is *dictionaries* that map keys (primitive values) to other values. A key-value pair is called an *entry*. We may consider all

$$
\begin{array}{lll}
\textit{update} & ::= & \textit{pupdate} \mid \textit{dupdate} \mid \texttt{void} \\
\textit{pupdate} & ::= & !\textit{primitive} \\
\textit{dupdate} & ::= & \{\textit{update\_entries}\} \mid \{\} \\
\textit{update\_entries} & ::= & \textit{update\_entry} \mid \textit{update\_entry}, \textit{update\_entries} \\
\textit{update\_entry} & ::= & \textit{primitive} \texttt{ -> } \textit{update}
\end{array}
$$

Figure 3: Syntax of updates

$$
\begin{array}{ll}
U[\![\texttt{void}]\!](v) & = v \\
U[\![!\textit{primitive}]\!](v) & = \textit{primitive} \\
U[\![\textit{dupdate}]\!](d) & = \begin{cases} U[\![\textit{dupdate}]\!](\{\}) & d \notin \textit{dictionary} \\ d' & d \in \textit{dictionary} \end{cases} \\
\multicolumn{2}{l}{\text{where } d' = \bigcup_{\forall k \in dom(\textit{dupdate}) \cup dom(d)} \{k\texttt{->}U[\![\textit{dupdate}.k]\!](d.k)\}}
\end{array}
$$

Figure 4: Semantics of updates

keys that do not exist in the definition are mapped to `null`. That is, `{"a"->null}` and `{}` are both empty dictionaries. The set of keys that are not mapped to `null` by a dictionary d is called the domain of the dictionary, denoted as $dom(d)$. We write $d.k$ for the value to which the dictionary d maps the key k. We also define a union operator on dictionaries, denoted as $d_1 \cup d_2$. The union $d_1 \cup d_2$ is a dictionary where each key k is mapped to $d_1.k$ if $k \in dom(d_1)$, otherwise is mapped to $d_2.k$.

There are many ways to map MOF-based models into Beanbag data. For example, one can encode a model into XML as defined in the XMI standard [13], and map the XML file (which is a tree structure and can be easily converted to dictionaries) into Beanbag format. We do not restrict a particular way of mapping models to Beanbag data, and users can choose their own way.

In this paper we map models into Beanbag data by assigning a unique identifier (ID) to each object, and an object reference is represented by the ID of the referred object. In a tool implementation, the IDs can be replaced by the in-memory addresses of the objects. The model is represented as a dictionary mapping IDs to the object instances. Each instance object is represented as a dictionary mapping feature names to feature values. For a multiple feature, we represent its value as a dictionary mapping IDs to the members in the feature (we currently do not consider the ordered features and the unique features). For each object, we also insert a special key `"__type"` mapping to the ID of its meta object in the meta model. For example, the `Display` class in Figure 1 is represented by the following dictionaries, where the numbers are generated IDs.

```
{1->{"__type"->20,
     "name"->"Display",
     "operations"->{10->2, 11->3, 12->4, 13->5}},
 2->{"__type"->21,
     "name"->"select",
     "parameters"->{},
     ...},
 3->{"name"->"stop",...},
 ...}
```

Updates Dictionaries not only allow us to represent many data structures but also enable us to uniquely identify each location in a dictionary. We make use of this feature to represent updates. An update in Beanbag is represented by the location of updates and the updated value. For example, if we rename the `Display` class to `DisplayWindow`, the update is described as: `{1->{"name"->!"DisplayWindow"}}`, where the last "!" indicates the original value is replaced by the new value following "!". Deleting an object can be represented as mapping the ID to `null` and inserting of an object can be represented as updates at a new ID. For example, deleting the `Display` class is represented as `{1->!null}`, and inserting a new class with a new ID 6 is represented as `{6->{"__type"->!20, "name"->!"NewClassName", ...}}`.

Figure 3 gives the syntax of updates. An update can be either *pupdate* – an update on primitive values, *dupdate* – an update on dictionaries, or void – indicating that nothing has been changed. An update on primitive values just contains a new value and an update on dictionaries maps keys to updates. If a key does not exist in the *dupdate* definition, we assume the key is mapped to void. The set of keys that are not mapped to `void` is the domain of the dictionary-update, denoted as $dom(\textit{dupdate})$. We also use the notation $\textit{dupdate}.k$, and $\textit{dupdate}_1 \cup \textit{dupdate}_2$ on dictionary-updates. Their meanings are the same as those defined on dictionaries.

Table 1: The result of $u_2 \circ u_1$

	$u_2 = \texttt{void}$	$u_2 \in \textit{pupdate}$	$u_2 \in \textit{dupdate}$
$u_1 = \texttt{void}$	void	u_2	u_2
$u_1 \in \textit{pupdate}$	u_1	u_2	u_2
$u_1 \in \textit{dupdate}$	u_1	u_2	u_3

where $u_3 = \bigcup_{\forall k \in dom(u_1) \cup dom(u_2)} \{k\texttt{->}(u_2.k \circ u_1.k)\}$

Figure 4 shows the denotational semantics of updates. The denotation of an update u, represented by $U[\![u]\!]$, is a function mapping between values. The denotation of void changes nothing. The denotation of *pupdate* maps any value to the new value, e.g., $U[\![\texttt{!3}]\!](2) = 3$. The denotation of *dupdate* applies every update in the dictionary-update to the value at the same key, e.g., $U[\![\texttt{\{2->!"a"\}}]\!](\texttt{\{2->"m"\}})=\texttt{\{2->"a"\}}$. One property of the semantics is that the denotation of an update is always idempotent, i.e., $\forall u \in \textit{update}, \forall v \in \textit{value} : U[\![u]\!](U[\![u]\!](v)) = U[\![u]\!](v)$.

Two updates can be merged (composed). Sometime users may perform a sequence of updates before the modeling tool can perform a fix, e.g., in a distributed environment. In such case we need to merge a sequence of updates into a single update. We use $u_2 \circ u_1$ to denote merging two updates u_1 and u_2 where u_1 is considered earlier than u_2. Table 1 shows the rules for merging two updates. For example, merging `{1->!"a", 2->!"b"}` with `{1->!"c", 3->!"d"}` results in `{1->!"c", 2->!"b", 3->!"d"}`. A requirement on merging is that merging should preserve the semantics of updates. In other words, $U[\![u_2 \circ u_1]\!] = U[\![u_2]\!] \circ U[\![u_1]\!]$. We can easily prove that the rules in Table 1 satisfy the requirement by checking the definitions.

However, if two users change the same location to different values, we say that the two updates by the two users conflict with each other. If two updates do not conflict, we say the two updates are compatible, denoted as $u_1 \oplus u_2$. We define compatibility from merging. Formally, $u_1 \oplus u_2$ iff $u_1 \circ u_2 = u_2 \circ u_1$. For example, `{1->!"a"}` and `{1->!"b"}` conflict but `{1->!"a"}` and `{2->!"b"}` are compatible.

A partial order can be defined over updates. We may want to know if one update u_1 is completely included in another update u_2. In other words, the locations changed by u_1 are all changed to the same values by u_2. This can also be formally defined by merging. If $u_1 \circ u_2 = u_2$, we say u_1 is included in u_2, denoted as $u_1 \sqsubseteq u_2$.

One property of Beanbag updates is that we can always find a minimal updates from a value v_1 to another value v_2

Table 2: The result of $find\_update(v_1, v_2)$ when $v_1 \neq v_2$

	$v_2 \in primitive$	$v_2 \in dictionary$
$v_1 \in primitive$	$!v_2$	$find\_update(\{\}, v_2)$
$v_1 \in dictionary$	$!v_2$	u

where $u = \bigcup_{\forall k \in dom(v_1) \cup dom(v_2)} \{k\texttt{->}find\_update(v_1.k, v_2.k)\}$

where all other updates from v_1 to v_2 include the update. We use a function *find_update* to get the minimal update from two values. Formally, $\forall u \in update, U[\![u]\!](v_1) = v_2 \Longrightarrow find\_update(v_1, v_2) \sqsubseteq u$. The function *find_update* can be defined as follows: 1) it returns `void` if the two values are equal, and 2) it follows the rules in Table 2 for other inputs.

4. CORRECT FIXING

Before explaining in detail how Beanbag describes consistency relations and fixes the inconsistency by update propagation, let us be precise about the properties a fixing procedure should satisfy.

Beanbag expressions often contain variables. To evaluate an expression or to propagate updates, we need to know what the current values of the variables are and what update users have performed on the variables. We use two sets of bindings to pass the information. The set of variable-value bindings, often denoted as σ : `VAR` $\rightarrow$ *value*, binds variables to data values. The set of variable-update bindings, often denoted as τ : `VAR` $\rightarrow$ *update*, binds variables to updates. We write $\texttt{var}^\sigma$ or $\texttt{var}^\tau$ for the value or the update bound to variable `var` in binding set σ or τ. We write $dom(\sigma)$ for the set of all variables in σ. We also write $\tau(\sigma)$ to denote applying all updates in τ to the corresponding variables in σ and returning a new set of variable-value bindings.

As we have seen in Section 1, a Beanbag program can be executed in either the checking mode or the fixing mode. We use two denotations to describe the semantics in the two modes. Suppose c is a constraint (an instantiated relation) defined by Beanbag. The checking denotation is a function which evaluates c according to a set of variable bindings.

$$E[\![c]\!] : (\texttt{VAR} \rightarrow value) \rightarrow \texttt{BOOLEAN}$$

For example, $E[\![\texttt{a=b}]\!]$ returns `true` for an input σ where $\texttt{a}^\sigma = \texttt{b}^\sigma$.

The fixing denotation is a partial function (called *fixing function*) that takes a set of value bindings and a set of update bindings and produces a new set of update bindings to satisfy the constraint.

$$R[\![c]\!] : (\texttt{VAR} \rightarrow value) \times (\texttt{VAR} \rightarrow update) \rightarrow (\texttt{VAR} \rightarrow update)$$

The function is partial (returning $\perp$ on some input) because the updates may conflict with each other or may not be allowed by the program. In such cases the modeling tool should report an error message to users. For example, given an input (σ, τ) where $\texttt{a}^\sigma = \texttt{b}^\sigma = 1$, $\texttt{a}^\tau = \texttt{void}$, and $\texttt{b}^\tau = \texttt{!2}$, $R[\![\texttt{a=b}]\!]$ returns τ' where $\texttt{a}^{\tau'} = \texttt{!2}$ and $\texttt{b}^{\tau'} = \texttt{!2}$. If $\texttt{a}^\tau = \texttt{!3}$ and $\texttt{b}^\tau = \texttt{!2}$, $R[\![\texttt{a=b}]\!]$ returns $\perp$.

Now we can turn to the correctness properties. First and foremost, we expect the fixing function to bring the data to consistency. The CONSISTENCY property requires that when we apply the output updates of $R[\![c]\!]$ to the input values, the values should be consistent according to $E[\![c]\!]$.

PROPERTY 1 (CONSISTENCY).
$R[\![c]\!](\sigma, \tau) = \tau' \Longrightarrow E[\![c]\!](\tau'(\sigma))$

Second, one naive way of achieving consistency is to simply modify the updated locations back to their original values. Such a naive fixing is not what we want. The PRESERVATION property prevents such a fixing by requiring the output updates to include the input updates. In this way, the output updates cannot change an updated location to a different value.

PROPERTY 2 (PRESERVATION).
$R[\![c]\!](\sigma, \tau) = \tau' \Longrightarrow \forall var \in dom(\tau) : var^\tau \sqsubseteq var^{\tau'}$

Third, if user updates do not actually change the model, (e.g., users choose to rename an operation but input exactly the same operation name), we do not want any part of the model to be changed. The STABILITY property ensures this by applying the output updates on the variables and check if the variables remain the same.

PROPERTY 3 (STABILITY).
$E[\![c]\!](\sigma) \wedge \tau(\sigma) = \sigma \Longrightarrow R[\![c]\!](\sigma, \tau)(\sigma) = \sigma$

There are two things to note about the properties. First, the properties define correct fixing in general, but satisfying the properties does not ensure the correctness of a particular fixing function. As a consistency relation may correspond to multiple fixing behaviors, the fixing function must ensure to take one that users want. Second, we do not require the input data bindings of a fixing function to be consistent. For example, it is possible $\texttt{a}^\sigma \neq \texttt{b}^\sigma$ in an input (σ, τ) to $R[\![\texttt{a=b}]\!]$. This is to support the "`or`" operator and the creation of new object, as we will see in Section 5.3.

5. THE BEANBAG LANGUAGE

Our Beanbag language is an OCL-like constraint language, which not only has usual checking semantics but also is equipped with a novel fixing semantics, which can systematically propagate updates through equal relations inherited in constraint description on both primitive values as well as dictionary structures.

5.1 An Overview

The left part of Figure 5 shows the syntax of the core Beanbag language.

The Beanbag language has similar syntax as OCL [12]. It has the primitive constraint "`=`" to describe equal relation between two variables, uses logic operators of `and`, `or` and `not`, and quantifiers of `forall` and `exists` on keys of dictionaries to construct complex constraints, and binds variables to expressions with the `let` construct. An expression may be a constant value, a dictionary key indexing $d.k$, or a local binding expression with `let`. With these constructs, Beanbag is powerful to describe various kinds of constraints; we have seen several examples in the introduction, and will see more examples in Section 5.4.

Different from OCL, the Beanbag provides the following declarative ways for people to define fixing behavior for reestablishing the consistency relation after an update happens.

- *Each standard constraint operator is equipped with a specific fixing operation.* For example, the primitive equation constraint "$v_1 = v_2$" will fix the relation by propagating updates from one to the other while treating v_1 with higher priority (so $v_2 = v_1$ has different fixing behavior from $v_1 = v_2$). The conjunction "c_1 `and` c_2" will

c	::=	`v=v`
	|	c `and` c
	|	c `or` c
	|	`d->forall(v|`c`)`
	|	`d->exists(v|`c`)`
	|	`d->exists!(v|`c`)`
	|	`let v=`e `in` c
	|	`protect v in` c
	|	`test` c
	|	`not` c
e	::=	*const*
	|	`d.k`
	|	`let v=`e `in` e
`v`	::=	any variable
`d`	::=	a dictionary variable
`k`	::=	a key variable

$$
\begin{array}{lcl}
E[\![\mathtt{v_1{=}v_2}]\!](\sigma) & = & \begin{cases} \mathtt{true} & {\mathtt{v_1}}^\sigma = {\mathtt{v_2}}^\sigma \\ \mathtt{false} & {\mathtt{v_1}}^\sigma \neq {\mathtt{v_2}}^\sigma \end{cases} \\
E[\![c_1 \ \mathtt{and}\ c_2]\!](\sigma) & = & E[\![c_1]\!](\sigma) \wedge E[\![c_2]\!](\sigma) \\
E[\![c_1 \ \mathtt{or}\ c_2]\!](\sigma) & = & E[\![c_1]\!](\sigma) \vee E[\![c_2]\!](\sigma) \\
E[\![\mathtt{d->forall(v|}c\mathtt{)}]\!](\sigma) & = & \forall k \in dom(\mathtt{d}^\sigma) : E[\![c]\!](\sigma[\mathtt{v} \mapsto \mathtt{d}^\sigma.k]) \\
E[\![\mathtt{d->exists(v|}c\mathtt{)}]\!](\sigma) & = & \exists k \in dom(\mathtt{d}^\sigma) : E[\![c]\!](\sigma[\mathtt{v} \mapsto \mathtt{d}^\sigma.k]) \\
E[\![\mathtt{d->exists!(v|}c\mathtt{)}]\!](\sigma) & = & E[\![\mathtt{d->exists(v|}c\mathtt{)}]\!](\sigma) \\
E[\![\mathtt{let\ v=}e\ \mathtt{in}\ c]\!](\sigma) & = & E[\![c]\!](\sigma[\mathtt{v} \mapsto E[\![e]\!](\sigma)]) \\
E[\![\mathtt{protect\ v\ in}\ c]\!](\sigma) & = & E[\![c]\!](\sigma) \\
E[\![\mathtt{test}\ c]\!](\sigma) & = & E[\![c]\!](\sigma) \\
E[\![\mathtt{not}\ c]\!](\sigma) & = & \neg E[\![c]\!](\sigma) \\
E[\![const]\!](\sigma) & = & const \\
E[\![\mathtt{d.k}]\!](\sigma) & = & \mathtt{d}^\sigma.\mathtt{k}^\sigma \\
E[\![\mathtt{let\ v=}e_1\ \mathtt{in}\ e_2]\!](\sigma) & = & E[\![e_2]\!](\sigma[\mathtt{v} \mapsto E[\![e_1]\!](\sigma)])
\end{array}
$$

Figure 5: Core Synatx and Checking Semantics

fix the relation using the fixing functions of both c_1 and c_2. The disjunction "c_1 `or` c_2" will fix the relation by first trying the fixing function of c_1 if c_1 is satisfied before updates happen, and that of c_2 otherwise, and if this fails we try the fixing function of the other. The forall qualifier "`d->forall(v|`c`)`" will fix the relation by fixing each dictionary entry with the fixing function of c if necessary.

- *New constraint constructors are introduced to describe different fixing functions.* Two forms of the existence constraint are provided to dealing with flexible fixing of dictionary structures. For instance, the two constraints

```
model->exists(class|class."name"=x)
model->exists!(class|class."name"=x)
```

describe the same consistent relation that there exist a `class` in the `model` whose name is equal to `x`. But they behave differently when the consistency is destroyed by, for example, a change of `x`. The former will create a new class with its name equal to the changed `x`, while the later will rename the existing class whose name is equal to `x` before `x` is changed.

- *New constructs are introduced to restrict fixing behavior.* The construct "`protect v in` c" describes the same constraint as c but does not allow its fixing function to update `v`, while the test construct "`test` c" describes the same constraint as c and allows no update on any variable.

In the following, after briefly explaining the common checking semantics, we focus on a detailed and formal definition of our new fixing semantics of the language.

5.2 Checking Semantics

The right part of Figure 5 shows the checking semantics of the Beanbag language. $E[\![c]\!](\sigma)$ and $E[\![e]\!](\sigma)$ evaluate a constraint c to a boolean value and an expression e to a value under a set of variable-value bindings, respectively.

We write $\sigma[\mathtt{var} \mapsto v]$ to indicate a new set of bindings that maps a variable `var` to value v and maps all other variables to the same values as σ. We will also use this notation for dictionaries and updates on dictionaries.

We can see that the checking semantics of Beanbag is the same as what we can expect from the syntax. In the following part we will focus on the fixing semantics.

5.3 Fixing Semantics

One of our major contributions is a natural and correct fixing semantics for Beanbag, an extended constraint language. Our idea is to propagate updates through equality constraints, control the propagation order by logic operators, derive structural updating through logic quantifiers, restrict fixing behavior through special constructs, and introduce recursion for describing more involved fixing strategies. We will use $R[\![\mathtt{c}]\!](\sigma, \tau)$ to describe the fixing for the constraint c under the variable-value binding set σ and an update described by the variable-update binding set τ. Its result is a new variable-update binding set showing how to update variables in such a way that c is satisfied again. We will define $R[\![\mathtt{c}]\!](\sigma, \tau)$ by induction on the construction of c.

Update Propagation based on Equality Constraints Propagating updates from one part to another to fix the inconsistency can be reduced to dealing with the following three equality constraints in our framework. This reduction will be explained in the fixing semantics for the `let` construct.

- $R[\![\mathtt{v_1{=}v_2}]\!](\sigma, \tau)$. Let us first consider a simple case, where the values of $\mathtt{v}_1$ and $\mathtt{v}_2$ are equal before updating τ, that is, ${\mathtt{v}_1}^\sigma = {\mathtt{v}_2}^\sigma$. In this case, we simply merge the input updates on $\mathtt{v}_1$ and $\mathtt{v}_2$ when they are compatible, and return $\perp$ when they conflict.

$$R[\![\mathtt{v_1{=}v_2}]\!](\sigma, \tau) = \begin{cases} \tau[\mathtt{v}_1 \mapsto u][\mathtt{v}_2 \mapsto u] & {\mathtt{v}_1}^\tau \oplus {\mathtt{v}_2}^\tau \\ \perp & \text{otherwise} \end{cases}$$

where $u = {\mathtt{v}_1}^\tau \circ {\mathtt{v}_2}^\tau$. More generally, the values of $\mathtt{v}_1$ and $\mathtt{v}_2$ may be unequal before updating, i.e., ${\mathtt{v}_1}^\sigma \neq {\mathtt{v}_2}^\sigma$. In this case, we first apply both updates to $\mathtt{v}_2$ to get a new value *new_v*, then calculate the update u needed to update $\mathtt{v}_1$ to *new_v*, and finally merge u with ${\mathtt{v}_1}^\tau$ and ${\mathtt{v}_2}^\tau$ to satisfy PRESERVATION.

$$R[\![\mathtt{v_1{=}v_2}]\!](\sigma, \tau) = \begin{cases} \tau[\mathtt{v}_1 \mapsto u][\mathtt{v}_2 \mapsto u] & {\mathtt{v}_1}^\tau \oplus {\mathtt{v}_2}^\tau \\ \perp & \text{otherwise} \end{cases}$$
$$\text{where } u = ({\mathtt{v}_1}^\tau \circ {\mathtt{v}_2}^\tau) \circ \mathit{find\_update}({\mathtt{v}_1}^\sigma, \mathit{new\_v})$$
$$\mathit{new\_v} = U[\![{\mathtt{v}_1}^\tau \circ {\mathtt{v}_2}^\tau]\!]({\mathtt{v}_2}^\sigma)$$

- $R[\![\mathtt{v}{=}const]\!](\sigma, \tau)$. To fix the equality constraint between a variable and a constant, we calculate an update over ${\mathtt{v}}^\tau$ by finding an update to change the updated `v` (i.e., ${\mathtt{v}}^{\tau(\sigma)}$) to the constant.

$$R[\![\texttt{v=}const]\!](\sigma,\tau) = \begin{cases} \tau[\texttt{v} \mapsto u \circ \texttt{v}^\tau] & u \oplus \texttt{v}^\tau \\ \bot & \text{otherwise} \end{cases}$$
where $u = \mathit{find\_update}(\texttt{v}^{\tau(\sigma)}, const)$

- $R[\![\texttt{v=d.k}]\!](\sigma,\tau)$. To fix the equality constraint between a variable `v` and a dictionary key indexing `d.k`, we first check the key `k`. If `k` is originally `null` with no update, denoted by $isNull(\texttt{k})$, we create a new key using the function $newID(\sigma,\tau)$ and do fixing. If `k` is deleted, we have no way to do fixing and return $\bot$. Otherwise, we do fixing on $R[\![\texttt{v=v}']\!](\sigma,\tau)$ where $\texttt{v}'$ is a fresh variable referring to the same value and update at $\texttt{k}^{\tau(\sigma)}$ in `d`. When $\texttt{v}'^{\sigma}$ or $\texttt{v}'^{\tau}$ changes, the $\texttt{d}^{\sigma}.\texttt{k}^{\tau(\sigma)}$ or $\texttt{d}^{\tau}.\texttt{k}^{\tau(\sigma)}$ changes accordingly. This is denoted by $R[\![\texttt{v=v}']\!](\sigma[\texttt{v}'\texttt{=d.k}^{\tau(\sigma)}], \tau[\texttt{v}'\texttt{=d.k}^{\tau(\sigma)}])$.

$$R[\![\texttt{v=d.k}]\!](\sigma,\tau) = \begin{cases} R[\![\texttt{v=d.k}]\!](\sigma, \tau[\texttt{k} \mapsto !newID(\sigma,\tau)]) & isNull(k) \\ \bot & \texttt{k}^\tau = \texttt{!null} \\ R[\![\texttt{v=v}']\!](\sigma[\texttt{v}'\texttt{=d.k}^{\tau(\sigma)}], \tau[\texttt{v}'\texttt{=d.k}^{\tau(\sigma)}]) & \text{otherwise} \end{cases}$$

Propagation Order Control based on Logic Operators We assign a fixing semantics to the logic operators of `and` and `or` to control the order of update propagation.

- The fixing function $R[\![c_1 \texttt{ and } c_2]\!]$ is to establish both c_1 and c_2. To do this, we call $R[\![c_1]\!]$ and $R[\![c_2]\!]$ one by one in this order to propagate updates. Since $R[\![c_2]\!]$ may propagate new updates to variables used in c_1, we may need to call $R[\![c_1]\!]$ again to satisfy c_1. Similarly, a call of $R[\![c_1]\!]$ may require a call of $R[\![c_2]\!]$. Hence in the fixing function we repeatedly call $R[\![c_1]\!]$ and $R[\![c_2]\!]$ until we reach a fixed point where no new update is propagated.

$$R[\![c_1 \texttt{ and } c_2]\!](\sigma,\tau) = \begin{cases} \tau' & \tau' = \tau \\ R[\![c_1 \texttt{ and } c_2]\!](\sigma,\tau') & \tau' \neq \tau \end{cases}$$
where $\tau' = R[\![c_2]\!](\sigma, R[\![c_1]\!](\sigma,\tau))$

  Beanbag does not always ensure the existence of such a fixed point. However, this will not be a problem in practice because most programs will terminate. We will discuss more on this issue in Section 6.

  It is worth noting a different order of c_1 and c_2 sometimes leads to different fixing behavior. We can write "c_1 `and` c_2" or "c_2 `and` c_1" to customize the behavior in such cases.

- The fixing function $R[\![c_1 \texttt{ or } c_2]\!]$ is to make either c_1 or c_2 be satisfied. As a result, in the fixing mode we can choose to use either $R[\![c_1]\!]$ or $R[\![c_2]\!]$ to propagate updates. However, to satisfy STABILITY, we must first use the one that is previously established on the data, otherwise the other constraint may change consistent data and violate STABILITY. For example, let us consider the constraint $R[\![\texttt{a=b."x" or a=b."y"}]\!]$. Suppose in the input data bindings `a` is equal to `b."y"` and the update bindings map both `a` and `b` to `void`. If we choose the first constraint, `a` will be changed to `b."x"` and STABILITY is violated.

  In $R[\![c_1 \texttt{ or } c_2]\!]$, we first find out the constraint that is previously established by calling $E[\![c_1]\!]$ and $E[\![c_2]\!]$, and use the constraint to propagate updates. If the constraint fails to propagate updates, we use the other constraint. Because here we switch from one constraint to the other, the data that are previously consistent for the former may not be consistent for the latter. That is why we require fixing functions to handle inconsistent data bindings. When the input data bindings are not consistent, we first try $R[\![c_1]\!]$ and then try $R[\![c_2]\!]$. This strategy is very useful in customizing the fixing behavior: programmers can assign a higher priority to a constraint by writing it first.

$$R[\![c_1 \texttt{ or } c_2]\!](\sigma,\tau) = \begin{cases} R[\![c_1]\!](\sigma,\tau) & \text{if } E[\![c_1]\!](\sigma) \wedge R[\![c_1]\!](\sigma,\tau) \neq \bot \\ R[\![c_2]\!](\sigma,\tau) & \text{elif } E[\![c_2]\!](\sigma) \wedge R[\![c_2]\!](\sigma,\tau) \neq \bot \\ R[\![c_1]\!](\sigma,\tau) & \text{elif } R[\![c_1]\!](\sigma,\tau) \neq \bot \\ R[\![c_2]\!](\sigma,\tau) & \text{otherwise} \end{cases}$$

Derivation of Structural Updating based on Logic Quantifiers The `forall` and `exists` quantifiers both relate an inner constraint to values in a dictionary. We will assign a fixing semantics to them to deal with updating on dictionary structures.

- The `forall` quantifier is satisfied only if the inner constraint is satisfied by all entries in the domain of the dictionary. Consequently we can call the fixing function of the inner constraint on all entries in the domain. One special case is deletion. Since we may need to propagate from the deletion of an entry, we also call on the deleted entries. However, we do not want to require the inner constraint to handle deletion, so we append "`or v=null`" to the end of the inner constraint. After invocation of the inner fixing functions, updates may be propagated to variables other than `v`, so we recursively call the fixing function until we reach a fixed point, the same as the `and` operator. In the definition we use an union operator on update bindings to construct the result. The union $\tau_1 \cup \tau_2$ is a set of bindings where the updates on the same variable in τ_1 and τ_2 are merged, and is $\bot$ when some updates conflict, or any of its operands is $\bot$.

$$R[\![\texttt{d->forall(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases} \tau' & \tau' = \tau \\ R[\![\texttt{d->forall(v|}c\texttt{)}]\!](\sigma,\tau') & \tau' \neq \tau \end{cases}$$
where
$\tau' = \bigcup_{\forall k \in D} . R[\![\texttt{c or v=null}]\!](\sigma[\texttt{v=d.}k], \tau[\texttt{v=d.}k])$
$D = dom(\texttt{d}^\sigma) \cup dom(\texttt{d}^{\tau(\sigma)})$

- The `exists` quantifier is satisfied if one entry in the dictionary satisfies the inner constraint. Therefore, we can fix an inconsistency by either 1) inserting a new entry in the dictionary that satisfies the inner constraint or 2) modifying an existing entry to satisfy the inner constraint. The `exist` quantifier chooses the first option. It generates a new key and invokes the inner constraint on the new key. Because the key does not in the domain of the dictionaries, `v` is initially mapped to `null` and `void` in the two binding sets. The inner constraint must change `v` to some value different from `null` otherwise the fixing function will return $\bot$.

$$R[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases} \tau & \text{if } E[\![\texttt{d->exists(v|}c\texttt{)}]\!](\tau(\sigma)) \\ R[\![c \texttt{ and not (v=null)}]\!](\sigma[\texttt{v=d.}k], \tau[\texttt{v=d.}k]) & \\ & \text{otherwise} \end{cases}$$
where $k = newID(\sigma,\tau)$

- The `exists!` construct explores the second option. It updates the entry that previously satisfies the inner constraint. When such an entry may not be found because the input value bindings may not be consistent. In this case it simply proceeds as `exists`.

$$R[\![\texttt{d->exists!(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases} R[\![c \texttt{ and not (v=null)}]\!](\sigma[\texttt{v=d.}k],\tau[\texttt{v=d.}k]) \\ \qquad \text{if } \exists k \in dom(\texttt{d}^\sigma) : E[\![c]\!](\sigma[\texttt{v} \mapsto \texttt{d}^\sigma.k]) \\ R[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma,\tau) \\ \qquad \text{else} \end{cases}$$

Restricting Fixing Behavior The constructs `protect`, `test` and `not` restrict their inner constraints from taking some fixing actions. These constructs are needed because sometimes we may want to reduce the fixing behavior. For example, it is possible that in `a=b`, `a` is considered as a source while `b` is considered as a read-only view where only source updates can be propagated to views and view updates cannot affect source. In this case we want to protect `a` from being modified by the fixing function of `a=b`.

- The "`protect v in` c" statement protects a variable from being modified by c. If c changes the variable, the `protect` statement will return $\perp$.

$$R[\![\texttt{protect v in } c]\!](\sigma,\tau) = \begin{cases} R[\![c]\!](\sigma,\tau) & U[\![\texttt{v}^{R[\![c]\!](\sigma,\tau)}]\!](\texttt{v}^\sigma) = \texttt{v}^{\tau(\sigma)} \\ \perp & \text{otherwise} \end{cases}$$

- The `test` construct protects all variables in the inner constraint. This construct is useful when we build an `or` constraint and we want to test some condition without changing anything.

$$R[\![\texttt{test } c]\!](\sigma,\tau) = \begin{cases} \tau & E[\![c]\!](\tau(\sigma)) \\ \perp & \neg E[\![c]\!](\tau(\sigma)) \end{cases}$$

- The operator `not` reverses a constraint. A constraint containing `not` is usually unfixable because we may face infinite choice of actions. For example, if "`not a=b`" is violated, we can change `a` and `b` to any pair of values that is not equal, and a fixing function cannot decide one. Nevertheless, `not` is still useful in testing conditions, so in Beanbag we define `not` in a similar way to `test`, where the fixing function simply returns $\perp$ when the constraint is not satisfied.

$$R[\![\texttt{not } c]\!](\sigma,\tau) = \begin{cases} \tau & \neg E[\![c]\!](\tau(\sigma)) \\ \perp & E[\![c]\!](\tau(\sigma)) \end{cases}$$

The `let` construct We have mentioned the fixing semantics of expressions can be reduced to an equality constraint in the "`v=`e" form. This reduction is done by the two `let` constructs when the constructs connect expressions and constructs together.

- The constraint "`let v=`e `in` c" is similar to "`v=`e `and` c" because it establishes the relations of both e and c. In the latter e becomes an equality constraint `v=`e. Since all expressions will eventually connect to a constraint by `let`, all expressions can be reduced to a equality constraint in this way.

  However, one difference between the above two constraint is that the `let` constraint has an inner variable `v` that initially has no bounded value. We must first set a proper value on `v` so that we can invoke the fixing functions of e and c. If e can be evaluated under the input value bindings, we produce the value by just evaluating e. If e cannot be evaluated (e.g., `k` is bound to `null` in `d.k`), we simply set the value of `v` to `null` to indicate an unknown value. After the value of `v` is properly set, we proceed to use the fixing function of `and`.

$$R[\![\texttt{let v=}e \texttt{ in } c]\!](\sigma,\tau) = \begin{cases} R[\![\texttt{v=}e \texttt{ and } c]\!](\sigma[\texttt{v} \mapsto val],\tau[\texttt{v} \mapsto \texttt{void}]) & val \neq \perp \\ R[\![\texttt{v=}e \texttt{ and } c]\!](\sigma[\texttt{v} \mapsto \texttt{null}],\tau[\texttt{v} \mapsto \texttt{void}]) & \text{otherwise} \end{cases}$$
where $val = E[\![e]\!](\sigma)$

- The statement "`let v=`e `in` e" will also be reduced to the "`v=`e" form and we define its fixing semantics using the previous `let` construct.

$$R[\![\texttt{v}_1\texttt{=(let v}_2\texttt{=}e_1 \texttt{ in } e_2\texttt{)}]\!](\sigma,\tau) = R[\![\texttt{let v}_2\texttt{=}e_1 \texttt{ in v}_1\texttt{=}e_2]\!](\sigma,\tau)$$

Recursion for More Involved Fixing Recursion is important to the description power of a language as it allows us to iterate over a recursive structure. Beanbag supports recursion by allowing us to define named constraints (called relations) and named expressions (called functions). We have seen `C1` and `C1onAll`, which are two examples of relations. Relations and functions can both be recursively called. For example, we can check if a class is not inherited from a particular class using the following code.

```
def check(class, parentRef, model) =
    test class.parent = null or
    (not class."parent" = parentRef and
     check(model.(class."parent"), parentRef, model))
```

5.4 Examples

In this section we give a few examples to show how to write Beanbag programs in practice. First, let us implement the same fixing behavior for relation C2 as in IBM RSA: 1) users cannot set the `sender`/`receiver` feature of a message to `null`, and 2) when a class instance in the sequence diagram is deleted, delete the connected messages. The Beanbag program is as follows.

```
def C2onAll(model) =
 model->forall(obj |
  isTypeOf(obj, "Message", metamodel)
   and not model.(obj."sender") = null
   and not model.(obj."receiver") = null
  or not isTypeOf(obj, "Message", metamodel)
  or obj = null)
```

The program uses `forall` to check all objects and within `forall` there are three constraints connected by `or`. The first constraint deals with `Message` objects and requires their `sender` and `receiver` features not to be `null`. The second constraint deals with non-`Message` objects and the third constraint deals with object deletion. The third constraint is actually included in the second, but it can take a fixing action (setting `obj` to `null`) while the second cannot.

When users try to change, for example, the `sender` feature to `null`, none of the three constraints is able to fix the inconsistency (the first two constraints have no fixing action to take and the last one cannot change `obj` to `null` because of PRESERVATION) and the fixing function will return $\perp$ to denote the update is not allowed. Now suppose users try to delete a class instance. When we visit to a message connected to the class instance, the first constraint will fail because the referred object is `null` and no fixing action can be taken. The second constraint will also fail because it has no associated fixing action. Finally, the third constraint will set `obj` to `null` to delete the message. In this way we can ensure all connected messages are deleted when a class instance is deleted.

The second example shows how to customize fixing behavior using `or`. Suppose we have a set of objects that may be persistent. If an object is persistent, it must be assigned to

a persistent container. As a result, when a persistent container is deleted, we may have two actions to take on the persistent object belonging to it; 1) we may delete the objects, or 2) we may simply change the `persistent` attributes of these object to `false`. The following program implements the first option.

```
def persistentConsistent(objs, model) =
 objs->forall(obj |
  obj."persistent" = true
   and not model.(obj."persistentContainer") = null
  or obj = null
  or obj."persistent" = false
   and obj."persistentContainer" = null)
```

This program has a similar structure to the first one. We use three constraints to deal with three different situations: the object is persistent, the object is deleted and the object is not persistent. When a persistent container is removed, the second constraint will delete the objects belonging to it. If we want to instead change the `persistent` attributes of these objects, we can just swap the last two constraints. After swapping, the priority of the attribute-changing constraint is higher than that of the object-deleting constraint and the fixing function will change the attribute rather than deleting an object.

Finally, let us construct the full program for relation C1. The program in Section 1 will always insert a new operation to resolve an inconsistency. However, if the inconsistency is caused by changing the receiver of a message, changing the base type of a class instance, or deleting an operation, we would prefer to set the name of the affected message to `null` to indicate that it does not relate to an operation. If users rename an operation in the class diagram, we would prefer to rename the related messages accordingly. The following program implements this fixing behavior.

```
def C1(msg, model) =
 let rec = model.(msg."receiver") in
 let opRefs = model.(rec."base")."operations" in
  protect model in
   (opRefs->exists!(r | msg."name"=model.r."name")
    and not msg."name" = null)
  or msg."name"=null
  or (opRefs->exists(r | model.r."name"=msg."name")
      and not msg.name = null)
```

This program connects three constraints using the `or` operator. The first constraint protects `model` so that updates are only propagated from operations to `msg`. The second constraint forces the message name to `null`. The last one is similar to the first but it does not protect `model`. When we rename an operation in a class diagram, the first constraint will propagate the update to the related messages. If we change the receiver of a message, change the type of a class instance, or delete an operation, the first constraint will fail because we cannot insert a new operation, and the second constraint will set the message name to `null`. If we rename a message to a new name, the first two will both fail and the third constraint will insert a new operation. In addition, we can still customize the fixing behavior of renaming a message by changing the last "`exist`" to "`exist!`".

6. DISCUSSION OF PROPERTIES

One important question to ask is whether the semantics of Beanbag satisfies the correctness properties that we have defined. The answer is positive. *Any Beanbag constraint satisfies* CONSISTENCY, PRESERVATION *and* STABILITY.

We can see this by using structural induction over the syntax rules for constraints. Most of the induction steps are straightforward, but there are two issues needed to be addressed. First, the `let` statement contains an expression and we need to know the properties of expressions before we discuss `let`. By using structural induction on expressions, we can see that expressions satisfy the following three properties, each corresponding to a property on constraint.

PROPERTY 4 (CONSISTENCY OF EXPRESSIONS).
$R[\![\mathtt{v}{=}e]\!](\sigma,\tau) = \tau' \Longrightarrow E[\![e]\!](\tau'(\sigma)) = \mathtt{v}^{\tau'(\sigma)}$

PROPERTY 5 (PRESERVATION OF EXPRESSIONS).
$R[\![\mathtt{v}{=}e]\!](\sigma,\tau) = \tau' \Longrightarrow \forall var \in dom(\tau) : var^{\tau} \sqsubseteq var^{\tau'}$

PROPERTY 6 (STABILITY OF EXPRESSIONS).
$E[\![e]\!](\sigma) = \mathtt{v}^{\sigma} \wedge \tau(\sigma) = \sigma \Longrightarrow R[\![\mathtt{v}{=}e]\!](\sigma,\tau)(\sigma) = \sigma$

Then we can reason the `let` statement using the above properties.

Second, several constraints and expressions use recursive calls to reach a fixed point. To satisfy STABILITY, we must ensure that such a fixed point always exists under the precondition of STABILITY. A fixed point exists if the function is increasing and has a upper bound. Because of PRESERVATION, the input updates must be included in the output. In this sense the fixing function is increasing. Because all inner constraints and expressions satisfy STABILITY, no variables will be changed when the precondition of STABILITY is satisfied. As a result, the updates on the variables cannot grow beyond the size of the bound values, and thus a fixed point always exists under the precondition of STABILITY.

Although the three properties are satisfied, it is possible that the fixing function of a Beanbag constraint does not terminate for some input when the precondition of STABILITY is not satisfied. For example, $R[\![$`a."x"=b and b."x"=a`$]\!]$ does not terminate for a input (σ,τ) where $\mathtt{a}^{\tau} = \mathsf{void}$ and $b = \{\}$. However, such a non-terminating Beanbag program often involves some counter-intuitive constraints (e.g., the example constraint is universally invalid) and is rarely encountered in practice. Based on our experience, *most Beanbag programs in practice always terminate.*

7. EVALUATION

Since Beanbag satisfies the correctness properties, in the evaluation we focus on the expressiveness and usability. We collected 32 consistency relations from the MOF standard [15] and 34 consistency relations on UML models from Alexander Egyed who used the relations to evaluate their fixing action generation work [3, 4][1]. From the 66 relations we identify 18 relations that can be automatically established through fixing actions. These relations are identified mainly through two criteria. 1) The fixing is sensible without human intervention or external information. 2) Fixing actions need to be taken. Some relations can be established without taking actions. For example, the name of a class should not be `null`. We can simply disallow users to change the name to `null`.

The selected consistency relations range from high level semantic relations like C1 to low-level syntactic relations like C2. For some relations, we also designed multiple fixing behaviors for each of them. As a result, we have the requirements for 24 Beanbag programs.

Then we proceed to implement these programs in Beanbag to see whether Beanbag is expressive enough for MOF

[1]In their publications they only mentioned 24 relations, but actually they have 34 relations in total.

and UML models. The result is encouraging. We have successfully implemented 17 programs, that is, about 71% of all programs. This result shows that *although Beanbag is not expressive enough for any fixing behavior, it can support many scenarios and is useful in practice.*

Reviewing the 7 unimplemented programs, we noticed that one program can be implemented with a trivial extension to Beanbag: a function counting the number of entries in a dictionary with no fixing action needed. The other 6 programs need a non-trivial, yet small extension to Beanbag: the ability to access the key when iterating entries in `forall`. This observation shows that *the problems on expressiveness are not fundamental.* All the 7 programs can be implemented by extensions under the basic philosophy of Beanbag: attaching fixing actions to primitive constraints and expressions, and composing them using high-level constructs.

On the whole, the development of a Beanbag program is much easier than manually implementing the fixing procedure. A Beanbag program is usually much shorter than a manually implemented fixing procedure, and Beanbag ensures CONSISTENCY, PRESERVATION and STABILITY of a program, which already eliminates many bugs.

However, during our development we also identified several problems on usability. First, Beanbag only ensures the correctness of the output updates, but does not ensure the existence of an output. It is up to the programmers to ensure the primitive constraints and functions are composed correctly so that the fixing function will not return $\perp$ for a proper input. As the interaction among constraints and expressions may be complex, it sometimes needs quite a few efforts to achieve this. Second, the fixing behavior involving inconsistent data is sometimes difficult to analyze. When we take inconsistent data into account, the domain of the fixing function becomes much larger. It is sometime very difficult to consider all situations. One possible solution to the two problems is to find some design patterns of Beanbag. We leave this for future work.

8. CONCLUSION

In this paper we have presented a novel language, Beanbag, for developing automated inconsistency fixing procedures on models. Beanbag attaches fixing actions to primitive constraints and functions, and composes them through logic operators and other high-level constructs. As a result, one Beanbag program has two meanings: one for defining a relation over data, and one for defining a fixing procedure that establish the relation over data by automatically propagating updates. Our study has shown that this approach greatly eases the development of fixing procedures and can support many, though not all, useful fixing scenarios in practice. Beanbag is implemented and is available on its website [22].

Acknowledgement

We gracefully acknowledge Alexander Egyed at the Johannes Kepler University for sharing us the UML consistency rules, all Beanbag users, especially Daniel Ruiz at the University of Malaga, for their creative programs and useful feedbacks, and all who commented on the early work of Beanbag.

9. REFERENCES

[1] M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *Proc. 2nd GTTSE*, pages 3–46, 2007.

[2] R. Balzer. Tolerating inconsistency. In *Proc. 13th ICSE*, pages 158–165, 1991.

[3] A. Egyed. Fixing inconsistencies in UML design models. In *Proc. 29th ICSE*, pages 292–301, 2007.

[4] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proc. 23rd ASE*, pages 99–108, 2008.

[5] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. 22nd ASE*, pages 64–73, 2007.

[6] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[7] J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania, 2005. Supersedes MS-CIS-03-42.

[8] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[9] T. Hettel, M. Lawley, and K. Raymond. Model synchronisation: Definitions for round-trip engineering. In *Proc. 1st International Conference on Model Transformation*, pages 31–45, 2008.

[10] N. Liu, J. Hosking, and J. Grundy. Maramatatau: Extending a domain specific visual language meta tool with a declarative constraint mechanism. In *Proc. VL/HCC*, 2007.

[11] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. 25th ICSE*, pages 455–464, 2003.

[12] Object Management Group. Object constraint language specification 2.0. `http://www.omg.org/spec/OCL/2.0`, 2006.

[13] Object Management Group. XML metadata interchange specification. `http://www.omg.org/docs/formal/07-12-01.pdf`, 2007.

[14] Object Management Group. MOF query / views / transformations specification 1.0. `http://www.omg.org/docs/formal/08-04-03.pdf`, 2008.

[15] OMG. MetaObject Facility specification. `http://www.omg.org/docs/formal/02-04-03.pdf`, 2002.

[16] D. Ruiz-Gonzalez, N. Koch, C. Kroiss, J.-R. Romero, and A. Vallecillo. Viewpoint synchronization of UWE models. In *Proc. 5th International Workshop on Model-Driven Web Engineering*, pages 46–60, 2009.

[17] A. Schürr and F. Klar. 15 years of triple graph grammars. In *Proc. 4th ICGT*, pages 411–425, 2008.

[18] H. Song, Y. Sun, L. Zhou, and G. Huang. Towards instant automatic model refinement based on OCL. In *Proc. 14th APSEC*, pages 167–174, 2007.

[19] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. 10th MoDELS*, pages 1–15, 2007.

[20] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. 6th UML*, pages 326–340, 2003.

[21] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[22] Y. Xiong. The Beanbag website. `http://www.ipl.t.u-tokyo.ac.jp/~xiong/beanbag.html`.

[23] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proc. 22nd ASE*, pages 164–173, 2007.